LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-512886

A Practical Mixed Analysis Framework for Arbitrary Binary Fragments

A. Saebjoernsen, R. Matzke, D. Quinlan

November 10, 2011

Architectural Support for Programming Languages and Operating Systems
London, United Kingdom
March 3, 2012 through March 4, 2012

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

A Practical Mixed Analysis Framework for Arbitrary Binary Fragments

[Anonymized for double-blind review]

ABSTRACT

We present ROSE, a high-fidelity, pluggable framework for binary analysis. ROSE has the unique capability of executing a program concretely or abstractly, starting from any offset. Thus, ROSE can analyze program fragments or functions in third-party binaries. The challenge inherent to starting from any offset is handling the initial lack of state. To overcome this challenge, ROSE accurately models each non-floating-point x86 instruction and how it operates on the heap, stack, and register file. This model and ROSE's support for system calls guarantee that no external, unanalyzable functions exist to limit or disrupt ROSE's analysis. ROSE supports execution over any abstract domain. Finally, ROSE integrates a disassembler and simulator: on behalf of the disassembler, the simulator resolves indirect jumps; in turn, the disassembler informs the simulator where valid instructions start. This synergism improves the state-of-the-art in detecting indirect jumps and functions.

To test ROSE, we compared its concrete interpretation against native execution, instruction by instruction, for programs in POSIX.1-2008. For system calls, including threading, ROSE marshals a program's interpreted state to and from native, concrete state; this mechanism passes the Linux Test Project. ROSE is easy to use: To incorporate a new abstract domain, one need only define a small set of functions. ROSE can simultaneously support multiple abstract domains and allow the user to determine which function it currently interprets. We use case studies to demonstrate its extensive feature set. ROSE is licensed under the MIT license and available for download at `<anonymized url>`.

1. INTRODUCTION

Binary analysis frameworks are essential. Many third-party applications and libraries are distributed without their source code. In addition, every program depends on external binaries, like the operating system and the source language's standard library and runtime system. Binary analysis is needed to thoroughly analyze such programs. Even when source code is available, due to compiler errors and optimizations, it is still desirable to analyze the executable code that actually runs on a system [2].

The loss of types and its simpler execution model distinguishes binary from source analysis. Existing binary analysis tools use approximations to recapture these abstractions. For instance, disassemblers are generally stateless and, as a result, can inaccurately guess how to parse object code into instructions. Most symbolic execution engines tether themselves to a concrete execution to resolve indirect jumps. A semantics-faithful simulation of a binary program, however, must precisely model the operation of each machine instruction on registers, stack and heap. We surveyed existing binary analysis tools and frameworks. Table 1 summarizes the most notable ones and their supported and missing features *w.r.t.* a careful selection of important desired functionalities.

1.1 Novel Combination of Features

Recognizing the important missing features from existing tools, our goal is to design and implement a binary analysis framework that can analyze a binary concretely, symbolically or abstractly from any offset and is equipped with a high-fidelity abstract model of storage. Our tool, ROSE, achieves its analytic power through its novel combination of features. Table 1 highlights the breadth and power of the features that ROSE possesses.

VM and Disassembler Integration All of the tools referenced in Table 1 are virtual machines, with the exception of CodeSurfer, which, at its core, is a disassembler[1] that builds a system dependence graph (SDG), the basis of many interesting static analyses, from its input. Like CodeSurfer, ROSE is also a disassembler and extracts control- and data-dependency from its input (although it does not yet build a SDG). ROSE is unique in integrating a virtual machine with a disassembler to improve the accuracy of both. The simulator resolves indirect jumps for the disassembler while the disassembler informs the simulator where valid instructions start.

High-Fidelity Abstract Storage Model One of ROSE's most distinguishing features is its high-fidelity abstract storage model. The model is critical for precise symbolic or abstract interpretation of binaries where concrete program states are often unavailable. One challenge that we have to overcome is the handling of system calls. ROSE computes values at various abstract storage locations through its various modes of execution. A high-fidelity storage model allows more concrete values to be computed for abstract storage locations. In essence, these concrete values shadow the operating systems values and obviate the slow and expensive simulation of the entire operating system. For system calls whose parameters are drawn from this subset, ROSE marshals their values into their native format needed for a system call's parameters, makes the native call, then unmarshals the result back into our abstract state. ROSE exits when it cannot concretize all of a system call's parameters. Users are free to define their own behavior here via ROSE's API.

Interpretation from Arbitrary Offsets In addition to improving the accuracy of our analyses, ROSE's rich and detailed storage model enables ROSE to analyze binary fragments from any offset. The key challenge to overcome to realize this feature is the lack of initial state. Concretely interpreting a program fragment using random values segfaults very quickly because generating a valid stack, heap and register file is highly improbable. Symbolic execution could be used to solve this problem, but it still requires either initializing symbolic state from a trace, which defeats the goal of starting from an arbitrary offset, or implementing symbolic storage, which treats registers, stack and heap locations as symbolic variables. The fact that ROSE can analyze binaries from any offset allows it to nicely

[1] It is built on top of IDA Pro [15].

Tools	Virtual Machine	Disassembler	PDG	Symbolic Execution	Abstract Interpretation	Pluggable Framework	High-fidelity Storage Model	Arbitrary Offset
Bochs	Yes	No	No	No	No	No	No	No
BitBlaze	Yes	No	No	Tethered	No	Yes	No	No
CodeSurfer	No	Yes	Yes	No	No	Yes	No	No
SAGE	No	No	No	Trace	No	?	No	No
QEMU	Yes	No	No	No	No	No	No	No
S^2E	Yes	No	No	Yes	No	Yes	No	No
Valgrind	Yes	No	No	No	No	Yes	No	No
ROSE	**Yes**	**Yes**	**Partial**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

Table 1: Feature comparison of binary analysis tools.

complement a source-level analysis like KLEE [7], which, instead of abandoning an execution path upon encountering an external function, could hand off the analysis to ROSE.

Mixed Interpretation In concrete execution, each variable has a single value drawn from its value domain. Concrete execution along a particular control-flow path yields a set of concrete variable-value bindings that drive program execution along that path. In symbolic execution, each variable assumes a subset of its value domain. Symbolic execution along a control-flow path maintains symbolic states (*i.e.* all possible values) of each program variable along that path; it therefore supports universally quantified reasoning about a path's state. Symbolic execution is not a panacea, however. It suffers from the typical path explosion problem (*i.e.* the fact that it may have to track states whose number can increase exponentially in the number of control points in the analyzed program) and theorem prover limitations (*i.e.* the fact that theorem provers, such as satisfiability modulo theorem (SMT) solvers, have difficulty reasoning about complex constraints). When symbolic execution is infeasible, concrete execution can be used to help make progress, at the cost of information loss. It is for this reason that ROSE, like its predecessors BitBlaze [27], S^2E [10] and SAGE [13], supports both concrete and symbolic execution.

Pluggable Abstract Interpretation Abstract interpretation is a powerful framework for static program analysis [11]. An abstract interpreter evaluates a program in an abstract domain; a well-chosen abstract domain is critical to obtain a precise and tractable analysis. ROSE is alone in supporting the abstract interpretation of binaries. Designing a good abstract domain is beyond the scope of this paper. That said, ROSE makes it convenient to implement abstract interpretation-based binary analyses: a programmer need only define at most a dozen functions that make the necessary changes to the semantics of the 100 x86 instruction that ROSE supports[2]. ROSE's abstract interpretation is tunable: it can abstractly interpret a program as much or little as desired, which is itself a function of how much the analyst knows about the program. A common tactic is to capture partial knowledge of how a program works in an abstract domain. ROSE can abstractly interpret a program in multiple domains at the same time. Finally, ROSE allows a programmer to add translators that can convert states between domains.

ROSE's extensible support for abstract domains is just one part of its general extensibility (Section 3.7). ROSE allows its user to change the semantics of an instruction, or a sequence of instructions, via adapters and callbacks. Its user can also change semantics of system calls, and she can change semantics of reading and writing memory. All these contribute to ROSE's ease of use for implementing binary analyses.

[2]These 110 instructions, approximately 17% of the x86 instruction set, are that subset of the x86 instructions necessary to support a subset of the POSIX.1-2008 [21] system utilities, chosen to exercise a wide variety of system calls, and the large programs, such as vim, that we have simulated with ROSE.

1.2 Contributions

This paper makes the following contributions:

1. The design and implementation of ROSE, an extensible binary analysis framework that features a high-fidelity abstract storage model and supports mixed (*i.e.* concrete, symbolic, and abstract) analysis of binary fragments from arbitrary offsets.

2. Empirical evaluation that shows the engineering quality of ROSE, including evaluating the correctness of its various components (such as instruction simulation and system call marshaling), its performance, and its capability for analyzing a binary from any offset.

3. Usage scenarios that illustrate ROSE's utility, including how to implement an analysis that can operate from any offset and how to use ROSE to aid debugging.

The rest of this paper is organized as follows. Section 2 uses a simple, classic example of abstract interpretation to illustrate ROSE's unique features and its ease of use. Section 3 presents ROSE's design and implementation, highlighting the important design choices and decisions. Section 4 first evaluates ROSE quantitatively to demonstrate its its quality and performance and then describes usage scenarios to illustrate its capabilities. Finally, we position ROSE against related work in the literature (Section 5) and conclude (Section 6).

2. ILLUSTRATIVE EXAMPLE

ROSE has a unique feature: it is capable of *mixed interpretation*, which allows a simulator to shift among or even simultaneously interpret a binary concretely, symbolically or abstractly. This section demonstrates ROSE's mixed interpretation and how ROSE is able to abstractly execute a code fragment, from an arbitrary offset, without resorting to a trace.

Binaries contain less information than source code — they lack types and have a low-level execution model, where, for example, a single program variable may be mapped into heap, stack or a register and that mapping can vary during a single run. Analyzing binaries for which one lacks source presents the challenge of handling this abstraction loss. When a compiler creates a binary, for example, it targets a particular machine when compiling the original source. Simulation attempts to interpret the binary as the targeted machine would, which can help recover some of the abstraction loss.

Each of ROSE's interpretation modes has different strengths. Concrete execution shines when a concrete memory state or trace is available. Loading, linking, and initializing (*i.e.* resources allocation such as libc initialization) a binary are three canonical examples. Concrete execution, however, cannot interpret code fragments from

```
class Analysis: public RSIM_Callbacks::InsnCallback {

virtual bool operator()(bool enabled, const Args &args) {
  if (args.insn->get_address()==simhalt_addr) {
    SymbolicSemantics::Policy symbolic_policy;
    X86InstructionSemantics<
      SymbolicSemantics::Policy, SymbolicSemantics::ValueType
    > symbolic_semantics(symbolic_policy);

    SignAnalysisExample::Policy sign_policy;
    X86InstructionSemantics<
      SignAnalysisExample::Policy,
      SignAnalysisExample::ValueType
    > sign_semantics(sign_policy);

    symbolic_policy.writeIP(
      SymbolicSemantics::ValueType<32>(analysis_addr)
    );
    while (symbolic_policy.readIP().is_known()) {
      uint64_t va = symbolic_policy.readIP().known_value();
      SgAsmx86Instruction *insn =
        args.thread->get_process()->get_instruction(va);
      symbolic_semantics.processInstruction(insn);
      sign_semantics.processInstruction(insn);
    }
  }
  return enabled;
}};

int main(){
  ...
  proj_addr_t main_addr =
    RSIM_Tools::FunctionFinder().address(project, "main");
  proj_addr_t ranfunc_addr =
    RSIM_Tools::FunctionFinder().address(project, "ranfunc");
  ...
  analysis analysis(main_addr, ranfunc_addr);
  sim.install_callback(&analysis);
  ...
}
```

Listing 1: Sign analysis in ROSE.

an arbitrary offset. Symbolic execution is particularly well-suited for interpreting code fragments with little or no state, or reasoning about all possible values of a particular variable. It is, however, inefficient and, in practice, is unable to analyze large program symbolically due to loops. Abstract interpretation enables reasoning about a subset of the properties of a program's state and, because it scales better than symbolic interpretation, can permit some analyses that would otherwise be infeasible.

Listing 1 shows a mixed analysis in ROSE that *concretely interprets* its input until the entry point of main where the simulation halts. The analysis resumes, *in symbolic execution mode*, at a specified arbitrary offset. The *abstract* sign analysis, a classic example of abstract interpretation [11], rides on top of the symbolic execution. Table 2 shows the abstract domain we use for sign analysis.

When a simulator only models program variables, the instruction pointer is concrete. ROSE completely models storage, which allows reasoning about variables in whatever storage, but complicates the handling of the instruction pointer. A naïve implementation of sign analysis in ROSE would abstract the instruction pointer into the sign domain and leave the simulator unable to determine the next instruction. Thus, Listing 1 again makes use of mix interpretation and runs its sign analysis on top of symbolic execution. By building the abstract analysis on top of a symbolic analysis, ROSE separates the concerns of discovering which instruction comes next into a concurrent symbolic analysis. Other than that they are decoupled;

Abstract Value	Semantic Interpretation
<+−0>	unknown (commonly denoted by $\top$)
<+−>	nonzero
<+0>	nonnegative
<−0>	nonpositive
<+>	positive
<−>	negative
<0>	zero
<>	undefined (commonly denoted by $\bot$)

Table 2: Abstract domain for sign analysis (with the natural ordering). Note that it is more fine-grained than the classic domain $\{\top, +, -, 0, \bot\}$.

```
1  int ranfunc(int seed) {
2    seed += (seed << 3);
3    seed ^= seed >> 11;
4    seed += (seed << 15);
5    return seed;
6  }
```

Listing 2: Input to the sign analysis.

ROSE is performing two analyses simulatiously in two distinct memories.

The example above first disassembles the program in order to find functions. On lines 25–26 we use the ROSE API to locate the starting addresses of two functions in the disassembled program, *viz.* the main function, at whose entry ROSE halts, and ranfunc where the analysis restarts. The class Analysis monitors the CPU instruction pointer. When it reaches the value of main, after dynamic linking has finished and the initial state is ready, it halts the simulation (line 4) and sets up the symbolic analysis to drive the sign analysis (lines 5–6). The symbolic analysis has its own heap, registers and stack, which associate a symbolic expression with each location. Since the desired domain is the sign domain, the analysis creates another memory and machine in the sign domain (lines 8–9).

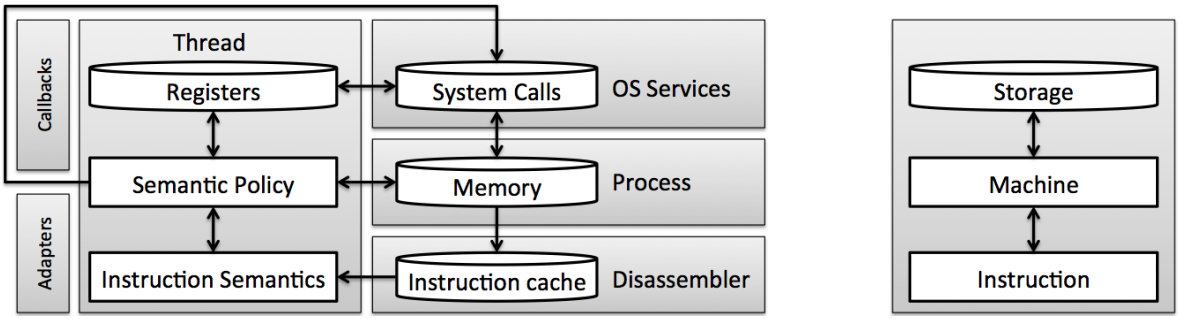
The symbolic execution restarts at an arbitrary offset, here the address of ranfunc (lines 11–12). It continues to the next instruction from this starting address until the next instruction is not unique (lines 13–19). Lines 14–19 perform the sign analysis on each instruction, where the symbolic analysis informs the analysis which instruction to execute next (line 18).

To perform sign analysis on ranfunc in Listing 2, one issues the command signanalysis ./ranfunc. Symbolic execution ends when ROSE cannot find a unique next instruction. The output lists all register and memory locations and their abstract values at the end of the analysis: *i.e.* ax = <+−0> (*i.e.* unknown), zf = <0> (*i.e.* zf is zero), sf = <+0> (*i.e.* nonnegative) and mem[+] = <+−0> (*i.e.* positive memory locations have unknown sign).

3. DESIGN AND REALIZATION

Our goal is to build a powerful, extensible, mixed interpretation binary analysis framework. There are many ways to tackle this ambitious goal. This section layouts the principal design choices and our design decisions, illustrated with examples drawn from our implementation.

ROSE is a simulator framework for 32-bit, Intel x86 [16] and AMD [1] Linux programs, and runs on 32 or 64-bit Linux. ROSE integrates the Yices SMT solver to reason about symbolic expressions [12]. Figure 1a depicts ROSE's architecture. This section discusses each of the components in the figure. The two core components we discuss first are cross-cutting concerns not explicitly



(a) The architecture of ROSE.

(b) The VM design space.

Figure 1: Overall design of ROSE.

present in the figure: ROSE's VM, which touches all the components, and ROSE's abstract storage model, which unites the register file and memory. Two core principles guided our design: 1) ease of use and 2) mix and match customizability.

A machine mediates the interaction of instructions and storage, as shown in Figure 1b. When using a VM to tackle binary analysis, one must decide how powerful and heavyweight to make the VM (Section 3.2), how to implement storage (Section 3.3) and whether and to what extent to allow manipulation of instructions (Section 3.4). To bootstrap interpretation, one needs a loader to read the specimen and its dependencies, like libraries, from disk and translate it into a form a VM can execute (Section 3.5). ROSE uses its disassembler to fetch instructions; Section 3.6 introduces it. ROSE is an extensible framework whose behavior can be modified in four ways: 1) instruction (re)definition, 2) system call (re)definition, 3) callbacks, and 4) adapters. Section 3.4 discusses instruction (re)definition; Section 3.7 discusses the last three mechanisms.

3.1 Terminology

The *specimen* is the program executing inside the simulator. We abbreviate operating system to OS and intermediate representation to IR. The *host OS* is the system for which ROSE was compiled and on which the simulator is running. The *guest OS* is the environment (system calls, signals, *etc.*) provided by the simulator to the specimen. Following convention, we use VM to denote virtual machine. We use *functor* to mean a C++ functor, or function object, not category-theoretic functor. We use *storage* to compactly refer to a machine's heap, stack and register file.

3.2 Virtual Machine

A VM must balance the performance of the specimen under emulation against the power of the analyses that it provides. Intuitively, there is a sliding scale from native execution to a complex abstract interpretation of a program. The core design decision the implementor faces, then, is to decide how to make this trade-off. Typically, VMs monitor program state and separate the specimen's state within the VM from the host OS. Recompilation-based VMs like Valgrind add the capability to manipulate instructions before executing them natively. Valgrind demonstrated how disassembling, instrumenting and recompiling super-blocks enabled more powerful analyses. Tools that find bugs in how a program manipulates memory are the most well-known applications of this capability.

In designing ROSE's VM, we embraced power, maximizing ROSE's ability to monitor and modify the specimen's behavior. In particular, ROSE is unique in abstracting storage. It is precisely this heavy-weight abstraction, described in Section 3.3 next, that, surprisingly, allows ROSE to reclaim performance. Essentially, abstract storage allows ROSE to lazily and abstractly define storage as determined by the needs of the instructions it is interpreting and therefore to begin interpretation from an arbitrary offset. Additionally, ROSE's support for abstract storage allows its user to inspect a specimen's entire storage state at any point in the simulation to determine, for instance, when to apply an expensive analysis, such as alias analysis. No other framework is capable of interpreting code fragments with little or no program state. An example application of this capability is unit testing for 3rd party proprietary libraries.

A direct consequence of this design decision is that ROSE simulates each instruction. For example, the X86InstructionSemantics class defines what basic operations must be performed by each x86 instruction; the Project_SemanticPolicy class defines the operations themselves. These classes cooperate to simulate the input specimen.

To illustrate the power of ROSE's feature set, we consider its per-instruction simulation in isolation. Simulating each instruction has a number of advantages over recompilation (*e.g.* Valgrind) or concrete interpretation. First, the specimen can be for a different architecture (guest) than that on which the simulator is running (host). Second, the simulator can handle privileged and normal instructions in a uniform way. Third, it is easy to modify the simulator to do something special for certain instructions. Fourth, it allows the simulator to keep the specimen in a separate address space that does not overlap with the simulator's own address space. Fifth, it provides a way for ROSE developers to gain confidence that ROSE's instruction semantics is working properly. The same instruction semantics is used for a wide variety of analyses within the ROSE library and in other tools. A bug in the implementation would likely cause the simulation to fail.

3.3 Abstract Storage

Storage systems and the instructions that operate on them must match: the instructions must read and write entities that the storage can contain; to turn it around, storage must contain entities on which instructions can operate. Achieving this match is the core design problem that abstract storage presents. To simplify this problem and to facilitate supporting many concrete ISAs, ROSE realizes its abstract storage in terms of its IR.

All existing VMs implement instruction semantics that operates on concrete program states. Unfortunately, concrete program states are sometimes unavailable. ROSE implements abstract storage to handle this case. When a concrete state is unavailable, ROSE can interpret instructions abstractly. This design decision means that ROSE is the machine, which obviates recompilation. Abstract storage contains the effect of an instruction's operation upon storage. Concrete memory maps virtual addresses to bytes. Symbolic or abstract memory must be defined in conjunction with symbolic or abstract instruction semantics in order to store sensible values and handle cases, such as unknown addresses or values.

```
template<size_t Len>
ValueType<Len> add(
    const ValueType<Len> &a, const ValueType<Len> &b
) const {
    if (a.sign==ZERO && b.sign==ZERO)
        return ValueType<Len>(ZERO);
    if (0==(a.sign & NEGATIVE) && 0==(b.sign & NEGATIVE))
        return ValueType<Len>(POSITIVE);
    if (0==(a.sign & POSITIVE) && 0==(b.sign & POSITIVE))
        return ValueType<Len>(NEGATIVE);
    return ValueType<Len>();
}
```

Listing 3: Redefinition of add for the sign domain.

```
case x86_lea: {
    if (operands.size()!=2)
        throw Exception("instruction must have two operands", insn);
    write32(operands[0], readEffectiveAddress(operands[1]));
    break;
}
```

Listing 4: Translating lea to internal IR.

To create a user-defined domain, a programmer need only redefine at most a dozen functions, not all the instructions in ROSE's IR. In ROSE's abstract storage, values are expression trees.

A domain is represented by a Policy class. Because each policy has its own distinct instance of memory, multiple policies can be used concurrently. To achieve this isolation, ROSE separates the specimen's address space from ROSE's. We see this in action in the illustrating example in Section 2 where three different memories are used concurrently; the program is interpreted concretely until the entrypoint of main where we initiate a symbolic interpretation of a code fragement. A sign analysis in the sign domain rides above the symbolic interpretation.

3.4 Instructions

The core problem one confronts when defining extensible instruction semantics is the other side of the matching problem that abstract storage presents: any tool that interprets program semantics must accurately model how instructions operate upon the program state. To support ROSE's rich and extensible abstract storage, ROSE's instruction semantics must operate upon abstract storage, of which concrete memory is a degenerate case, to support analyses of code fragments at an arbitrary offset. Consonant with our core design principle that ROSE be extensible, ROSE allows a developer to change or manipulate instruction semantics at two stages: the developer can change either 1) how an interpreted instruction is translated into our RISC-like instruction set or 2) how ROSE interprets an instruction in its IR, as, for instance, redefining the domain that instruction operates upon. Listing 3 shows the redefinition of the add instruction to support sign analysis.

Most VMs or emulators choose to translate between the interpreted instruction set (*e.g.* x86) and an IR in a RISC-like instruction set to simplify manipulating instruction semantics and reduce the chance of implementation bugs. ROSE is no exception to this rule. Listing 4 shows how compactly translates an emulated instruction into its IR: a simple instruction has a simple translation.

ROSE currently supports 110 instructions[3], about 17% of the total in the x86 instruction set [30]; it treats unsupported instructions as no-ops, since this allows ROSE to ignore instructions that provide inessential functionality. Broadly, ROSE does not support the following categories of instructions: MML, 64-bit, floating-point, MMX, 3DNow!, SSE, SSE2, SSE3, SSE4, SSE5, AVX FMA, and AES. ROSE also does not, in general, simulate instructions that manipulate control registers (CR0-4 and CR8), debug registers (DR0-3 and DR6-7), test registers (TR3-7), descriptor registers (GDTR, LDTR, IDTR), task register (TR), or model-specific registers (MSRs). ROSE actually does simulate a handful of instructions from these classes, such as the MOVD, MOVQ and two MMX instructions, but only those required by the dynamic linker and glibc.

With the exception of instructions listed in one of the above categories, such as x686 instructions, ROSE *does* simulate all integer instructions in x86 up to the 386 instruction set.

[3]Section 4 explains how these 110 instructions were chosen.

3.5 Loader

A loader reads a specimen and its dependencies, like libraries, from disk and translates it into a form a VM can execute. BinaryLoader implements ROSE's loader; it defines the public interface and provides generic implementations for loading — parsing, linking, mapping, and relocating — a static or dynamic object. Parsing reads a binary file and parses its container format (ELF, PE, COFF, Dwarf, *etc.*) to produce an abstract syntax tree (AST); it does not parse the machine instructions, *viz.* disassembly is not part of loading. Linking recursively parses all shared object dependencies. Mapping chooses virtual addresses for parts of the binary file as if ROSE were creating a new OS process. For instance, mapping an ELF file causes ROSE to choose virtual addresses for all ELF segments. Relocation applies relocation fixups to patch pointers and offsets in various parts of the virtual address space.

ROSE's loader was designed to load a binary into memory as an operating system would, supporting both static and dynamic executables. It is cross-platform, supporting a variety of binary container formats (ELF, PE, *etc.*) and loaders (Linux and Windows). To ROSE's linking to emulate linking of one operating system while Project is running in another operating system, the linker can control the locations of shared object dependencies without affecting how Project itself is loaded. ROSE's linker can partially load an object and thereby handle the case that not all libraries are available. As with ROSE's other components, its loader is user-extensible: user can register their own loader or change the behavior of a loader at runtime

3.6 Disassembler

The Disassembler is a modular and extensible API providing all non-architecture-specific functionality for recursively disassembling instructions and grouping them into basic blocks. The disassembler currently supports the x86, ARM and PowerPC architectures. Binary executables are often dynamically linked, wherein the main binary contains only stubs for dynamic functions and those functions' text and data are loaded into process memory and linked at run time. Unlike most disassemblers, ROSE's disassembler performs these steps when disassembling, and can therefore disassemble the dynamic libraries. The disassembler can be used in conjunction with or separately from the simulator. It is user-extensible; the user has only to define its architecture-specific components to support a new architecture.

ROSE's uses disassembly to fetch one instruction at a time, caching the instruction as it receives them. Users can add analyzes to the simulator that use the disassembler. For instance, the disassembler can extract extra information by performing a control flow analysis of the function that is currently executing.

The simulator can control which memory the disassembler reads and disassembles. Unpackers are a classic example: ROSE can simulate the unpacker and, when the unpacker is about to jump into the newly unpacked instructions, invoke the disassembler. The

simulator can also provide the disassembler with a linked version of a program for disassembly.

3.7 Customization

The challenge is to provide customization in a principled fashion. The issues any solution to this challenge must consider include thread-safety, how much customization to allow (should a subset of the simulator's behavior be inviolate?), how do customization compose/interact and whether to support dynamic customization. ROSE is relentlessly customizable; it is thread-safe; it supports compositional and dynamic customization. ROSE provides four mechanisms to realize customization with these features — instruction (re)definition, system call (re)definition, callbacks, and adapters. Instruction definition is discussed above in Section 3.4. Below, we discuss how ROSE's system call mechanism, although required by abstract storage, doubles as a customization mechanism that allows a user to augment or replace system calls, by adjusting ROSE's system call table. Many of the events that occur during the simulation can be tied to user-defined callbacks invoked before and/or after the action. The callbacks augment, replace, or skip the normal processing depending on the situation. For instance, a pre-instruction callback can check for unsupported instructions and skip them, or a pre-system-call callback can count the total number of system calls executed per thread. Adapters modify the behavior of the simulator. For instance, if you need a simulator that prints hex dumps of all data transfered over a TCP network connection, you would attach an Project_Adapter::TraceTcpIO adapter to the simulator.

System Call Marshaling

ROSE's abstract storage necessitates special-handling of system calls. The problem is translation from a specimen's internal, arbitrarily abstract state running under ROSE into the concrete parameters that a system call requires. To solve this problem, ROSE marshals the state necessary to perform a system call to and from a specimen's abstract storage. Currently, ROSE only performs a system call if enough state exists in a specimen's abstract storage to provide the native system call with its arguments and exits otherwise, since system calls are almost always essential. If enough state does not exist, the user must either treat the system call as a no-op or provide an alternate mechanism for constructing the necessary state.

ROSE is designed to simulate a specimen such that its abstract storage is equivalent to the same abstraction applied to the concrete memory the specimen produces when running natively. The fidelity of ROSE's marshaling enables the specimen to interact with the host OS and programs running natively through system calls. ROSE handles the following categories of system calls: file system, inter-process communication, memory management, memory maps, process properties (*e.g.*, getuid, setpgid, setrlimit), signals, socket calls, standard I/O, threading and time.

Listing 5 shows how to redefine sys_chown (#182) to return the "not implemented" error. To completely remove a system call implementation and cause the simulator to dump the specimen's core if it tries to invoke that system call, one has only to remove all enter, body, and leave callbacks for that system call. In Listing 6, we undefine sys_fork (#2).

```
1  class NoOp: public RSIM_Simulator::SystemCall::Callback {
2  public:
3    bool operator()(bool b, const Args &args) {
4      args.thread->tracing(TRACE_SYSCALL)->more(
5        "[NOOP]"
6      );
7      args.thread->syscall_return(-ENOSYS);
8      return b;
9    }
10 } syscall_noop;
11
12 RSIM_Simulator *simulator = ...;
13 simulator->syscall_implementation(182)->body.clear().append(
14   &syscall_noop
15 );
```

Listing 5: Redefining a system call

```
1  RSIM_Linux32 *simulator = ...;
2
3  RSIM_Simulator::SystemCall *sc_fork =
       simulator->syscall_implementation(2);
4  sc_fork->enter.clear();
5  sc_fork->body.clear();
6  sc_fork->leave.clear();
```

Listing 6: Undefining a system call

Callbacks

A callback is a user-supplied objects whose **operator**() is invoked at particular points during a simulation. Callback points are associated with threads, processes, or the simulator as a whole. ROSE's callbacks are a convenience mechanism focused on easing the writing of analyses. For instance, a disassembler can be invoked to look-ahead at future instructions before they are actually executed.

Callbacks are organized into lists. When the simulator reaches a callback point, it invokes all the callbacks present in that point's list. When a new thread is created, its callbacks are initialized from the process; when a new process is created, its callbacks are initialized from the simulator. Removal of all callbacks from the list at a callback point does not delete that point. The arguments for a callback depend on the category of its callback point, but always include a boolean value, which is the return value of the previous callback on the list (or related list), or true for the first callback. Callback are thread-safe.

Listing 7 shows the definition of a callback that prints "[FIRST CALL]" the first time a particular system call executes, then removes itself on line 10. To bind this callback to system calls 3 and 4, the programmer writes

```
1  RSIM_Linux32 *simulator = ...;
2  NotifyOnce notifier;
3  simulator->syscall_implementation(3)->enter.append(&notifier);
4  simulator->syscall_implementation(4)->enter.append(&notifier);
```

Listing 8 shows a callback that implements instruction granular traces. In ROSE's AST, the function that contains an instruction is the instruction's grandparent; the callback leverages this fact. The developer attaches this callback to the simulator with

```
sim.get_callbacks().add_insn_callback(
    RSIM_Callbacks::BEFORE, new ShowFunction );
```

and a snippet of its output is

```
28129:1 0x0805e7c0[256]:   in function "__uname"
28129:1 0x0805e7cd[260]:   uname[122](0xbfffddb6) = 0
...
28129:1 0x0804c375[348]:   in function "__libc_setup_tls "
28129:1 0x0805f5c8[376]:   in function "__sbrk"
28129:1 0x080871f0[387]:   in function "brk"
28129:1 0x080871fe[393]:   brk[45](0)  = 0x080d5000
```

The "28129:1" means the main thread of process 28129. The hexadecimal number is the address of the executed instruction followed by the value of the instruction counter in square brackets. The "in function" output includes the system call along with its arguments and return value.

```
class NotifyOnce :
  public RSIM_Simulator::SystemCall::Callback {
public:
bool operator()(bool b, const Args &args) {
  args.thread->tracing(TRACE_SYSCALL)->more(
    "[FIRST CALL]"
  );
  RSIM_Simulator *sim =
    args.thread->get_process()->get_simulator();
  sim->syscall_implementation(args.callno)->enter.erase(this);
  return b;
}};
```

Listing 7: Callback that prints first use of a system call.

```
class ShowFunction: public RSIM_Callbacks::InsnCallback {
public:
  virtual bool operator()(bool prev, const Args &args) {
    SgAsmBlock *basic_block =
      isSgAsmBlock(args.insn->get_parent());
    SgAsmFunctionDeclaration *func =
      basic_block ?
      SageInterface::getEnclosingNode<
        SgAsmFunctionDeclaration
      >(basic_block) : NULL;
    if (func && func->get_name()!=name) {
      name = func->get_name();
      args.thread->tracing(TRACE_MISC)->mesg(
        "in function \"%s\"", name.c_str()
      );
    }
    return prev;
  }
private:
  std::string name;
};
```

Listing 8: Instruction granular tracing.

Adapters

An adapter is an object that changes a simulator's behavior. For instance, adapters can be used to change the semantics of instructions or system calls. Adapters can be attached, even stacked, and detached to restore the simulator to its original behavior. ROSE's adapters are thread-safe. TraceFileIO is an example adapter that monitors data transfers on specified file descriptors and reports them using the tracing facility. To accomplish its task, TraceFileIO registers additional callbacks on the relevant system calls. Users could manually augment system calls, but using an adapter is a good way for the user to ensure that all system calls relevant to a certain kind of analysis are properly augmented. Listing 9 shows how to use TraceFileIO.

```
Project_Linux32 simulator;
Project_Adapter::TraceFileIO tracer;

tracer.trace_fd(0); // standard input
tracer.trace_fd(1); // standard output
tracer.trace_fd(2); // standard error

tracer.attach(&simulator);
...
tracer.detach(&simulator);
```

Listing 9: Adaptor example.

4. EVALUATION

We evaluate ROSE along two dimensions. First, we evaluate the correctness of its implementation of instruction semantics and its system call marshaling mechanism, then present ROSE's performance when used merely as a concrete simulator. Second, we describe how to use ROSE to jump to an arbitrary offset, then present a collection of debugging scenarios. The arbitrary offset scenario shows how, especially for a large program like vim, ROSE's support for interpreting from an arbitrary offset mitigates its simulation overhead. The evaluations described herein ran on a 12-core Intel Xeon X5680 3.33GHz with 48GB per core running Linux 2.6.32-5-amd64.

4.1 Correctness and Performance

	Total	Simulator	Disassembler
System calls	293	110	–
Instructions	663	115	663

Table 3: Number of system calls and instructions whose semantics under ROSE's simulator and disassembler were verified.

Table 4 shows how many of the total system calls and instructions we have implemented. System calls do not make sense for the disassembler as interrupts trigger them. In this section, we focus on verifying the simulator, since verifying the semantics of the disassembler consists solely in matching its output against the table of opcodes documented in architecture specifications.

ROSE implements all integer instructions that operate on integer registers in the 386 architecture, except those instructions introduced in 386 that operate on control and debug registers, such as dr0 or dr1. POSIX.1−2008 [21] specifies 161 system utilities, and is part of the Single UNIX Specification (SUS). These utilities can be found on most UNIX-like operating systems. These utilities interact with and change system state using system calls, and are likely to exhibit more complex system call behavior than most programs. Because of this we decided to implement the necessary functionality to support these utilities one-by-one. We currently support 59 of these utilities. The 110 instructions ROSE current supports as precisely those needed to run these 59 utilities. We believe the instruction usage patterns of these utilities to be representative and that ROSE can already support wide variety of applications. Development of ROSE is ongoing; support for additional instructions is added as needed.

Verifying* ROSE*'s Interpretation of Instructions A simulator must correctly implement instruction semantics, which are strange, complicated creatures. The core idea is to run ROSE in parallel with a native execution of the executable ROSE is analyzing and compare the input-output results of each executed instruction. The specimen is run under a very simple network-based debugger which, in turn, runs under ROSE.

To ease this task, we implemented a semantic analyzer. This tool seeks to verify ROSE's interpretation of an instruction's semantics accurately reflects how the instruction executes on real hardware. The semantic analyzer queries the debugger for value of every register and memory read operation and stores the result.

ROSE parses the executable and obtains the addresses of all executable segments. It then sets breakpoints for all addresses in those segments and asks the debugger to continue execution. When a breakpoint is reached, ROSE looks up the SgAsmInstruction at that address. ROSE queries the debugger for the instruction, then submits it to the semantic analyzer. ROSE asks the debugger to single step and then compares writes, from previous step, with values queried from the debugger. ROSE and the debugger then advance to the next breakpoint and repeat the process.

This evaluation methodology is conservative. First, the verifier does not handle interrupts. Therefore, instructions like "INT" will

	Total Verified	Errors Reported	Actual	CI
diff	454822	490	0	22.84
grep	470688	1728	0	23.35
gzip	235550	298	0	7.54
ls	454760	614	0	22.99
patch	475613	1048	0	23.22

Table 4: Verification of instruction semantics: Confidence level is set to 99% and except for gzip the sample size is 30. For gzip we analyzed every single reported error. 50% percentage was used to calculate the confidence interval.

execute in the natively-running version (under the debugger) but not in the ROSE's simulator. Therefore, one should expect the state after interrupts to be different. Second, instructions that interact with the environment, like RDTSC (read timestamp counter) return different values in different executions, such as native and simulated runs.

Since the goal of evaluation is instruction coverage, not specimen path coverage, so we use a minimal input. For instance, the gzip test is executed as:

```
echo > gzip.input
gzip < gzip.input >/dev/null
```

We began our evaluation by manually inspecting every discrepancy between the native and simulated output reported by the semantic analyzer for gzip. There were 498 such failures, 183 of which are due to system calls. The verifier does not handle system calls, so state always mismatches and generates a failure after an "INT 0x80" (183 calls). Likewise, there are 17 failures due to SYSENTER. After eliminating these two sources of failure, 298, the number reported in Table 4, remain. In addition to the system call and SYSENTER errors themselves, the instruction that follows them also generates an error, resulting in 200 errors. For those cases (approximately 50) we inspected, the only difference in state is the "ORIG_EAX" pseudo-register not accounted for by our semantic analyzer. This is expected, and we can ignore this failure. Of the remaining 98 errors, 48 failures are in the simulation of the MOV instruction. The EIP register had a simulated value of 0x40000ff7 after the instruction, but the debugger reported 0x40000ff0. In this case, the debugger is wrong because the instruction was at 0x40000ff0, so the EIP point to an address after it. The final 50 failures occur because the verifier currently does not support discovering the base address for the GS register. Basically, a memory reference is something like GS:[offset], where GS is a segment register and "offset" is a memory address, like 0x08040123. Segment shadow registers are internal to the x86 CPU — they are not externally accessible.

After our thorough manual inspection of one program, we then manually inspected 30 reported failures, chosen uniformly at random, to verify the instructions of the remaining programs. Table 4 reports the resulting interval into which the actual error falls with 99% confidence.

System Calls The Linux Test Project (LTP) [18] has the goal to deliver test suites to the open source community that validate the reliability, robustness, and stability of Linux. The LTP test suite contains a collection of tools for testing the Linux kernel and related features. LTP tests both passing and failing system call behavior. ROSE uses LTP to test its system call marshaling; thus, ROSE uses, and passes, exactly the same test suite used to test the system call implementation of the Linux kernel itself. To ensure that ROSE stays conformant, we use continuous integration to enforce that all versions of ROSE pass these unit tests.

For threading, we used the pthread conformance tests in the POSIX Test Suite to perform conformance, functional, and stress testing of the IEEE 1003.1-2001 System Interfaces specification in a manner that is agnostic to any given implementation. We apply the unit tests of all 104 pthread routines.

Table 5 reports 9 pthread unit test failures. One of the tests, thread_create.10−1.c, exhibited nondeterministic behavior both natively and under ROSE. Four pthread_mutexattr tests did not compile, generating undeclared variables errors. The pthread_once test had no main function. Three tests fail when run natively, outside the simulator, within 2 minutes.

There are 293 system calls in Linux Kernel 2.6.7; we currently support 110 of those system calls via marshaling. IBM has contributed a system call test suite to the Linux Test Project with 339 test programs with units tests that test the 110 system calls we support. In those 339 files the total number of passing test conditions is 421 and total number of failing test conditions is 1295.

The sys_gettimeofday test sometimes fails both natively and under ROSE because the kernel has a bug in the way it interacts with the RTC that causes the clock to go backward by small amounts every so often[4]. The single test of sys_fchown16 does not deterministically pass natively on some systems and we should therefore not expect it to pass under ROSE either. Four tests of sys_ipc and two tests of sys_fcntl64 fail because our implementation of the futex() system call is incomplete.

Performance Table 6 compares ROSE when concretely simulating the selected programs without jumping to an arbitrary offset. The completion times are real, elapsed time. The ROSE simulator was an optimized version, compiled using "−O3 −fomit−frame−pointer". The specimens are all dynamically linked, 32-bit x86 ELF binaries.

Table 6 reports the total number of system calls to demonstrate the fidelity of ROSE's simulation. Since there is no equivalent tool to ROSE, we have evaluated it against Valgrind, a powerful, publicly available binary analysis tool. We do not report the system calls Valgrind executes because Valgrind is not natively aware of system calls. Currently, ROSE shares its standard out with its specimen. As a result, ROSE does not allow a specimen to close stdout. Some programs respond by retrying to close stdout. Another source of discrepancy is when a program calls execve to start; ROSE ignores execve because when ROSE is running, its specimen has already started. ROSE does not support floating-point, which affected the patch program. In spite of this fact, ROSE's simulation produced correct output. The unique system call column reports the number of unique system calls made during the run. We did not report this number for Valgrind, again because Valgrind does not natively support collecting such data. Table 6 similarly reports the total number of instructions executed, which number in the millions, as well as the unique instructions whose repeated execution generated that total.

Table 6 shows that ROSE is slower than Valgrind, when fully simulating a program. This is not surprising given its greater power. There are three reasons for this slowdown. The first source of slowdown is that ROSE disassembler is used to build an intermediate representation of each instruction before the instruction can be simulated. Disassembly at some level is necessary in order to simulate instructions, and the level of disassembly that the simulator uses is the same as that which ROSE uses for all other binary analyses. The second source of slowdown is that dynamic linking in the specimen is resolved as a side-effect of simulation. The simulator loads the specimen's executable into an address space (MemoryMap) just as the host OS would, and then begins simulation. If the main

[4] http://www.gossamer-threads.com/lists/linux/kernel/813344.

	Total Number	Number Supported	Number of Tests	Tests passing natively	Tests passing	Tests failing
System calls	293	110	339	337	331	8
pthread routines	104	104	417	408	408	9

Table 5: Statistics about instructions and system calls supported by ROSE.

		completion time	System Calls total	System Calls unique	Instructions total	Instructions unique
gzip	native	4ms	203	17	5.7m	–
	Valgrind	0.4s	–	–	5.7m	–
	ROSE	9.4s	203	15	5.7m	73
diff	native	3ms	260	21	2.3m	–
	Valgrind	0.5s	–	–	2.3m	–
	ROSE	4.1s	264	21	2.3m	78
patch	native	4ms	361	26	4.6m	–
	Valgrind	0.5s	–	–	4.6m	–
	ROSE	8.0s	356	25	4.6m	85
grep	native	3ms	205	14	1.0m	–
	Valgrind	0.5s	–	–	1.0m	–
	ROSE	1.8s	208	14	1.0m	73
sed	native	5ms	265	15	12.9m	–
	Valgrind	0.5s	–	–	12.9m	–
	ROSE	22.1s	266	14	12.9m	77
ls	native	3ms	338	26	0.9m	–
	Valgrind	0.5s	–	–	0.9m	–
	ROSE	1.6s	338	24	0.9m	73

Table 6: ROSE full simulation performance results.

```
unsigned long updcrc(const unsigned char *s, unsigned n)
{
    unsigned long crc_32_tab[256];
    register unsigned long c;
    unsigned long crc = (unsigned long)0xffffffffL;
    ...
    // huge assignment to crc_32_tab
    ...
    if (s==NULL) {
        c = 0xffffffffL;
    } else {
        c = crc;
        if (n) do {
            c = crc_32_tab[((int)c ^ (*s++)) & 0xff] ^ (c >> 8);
        } while (--n);
    }
    crc = c;
    return c ^ 0xffffffffL;
}

/* The simulation stops here. */
int main() { return 0; }
```

Listing 10: Input used to evaluate arbitrary offset.

executable has an interpreter, then the interpreter is also loaded and simulation starts in the interpreter. In this way, the simulator is able to simulate dynamically linked executables; it simulates the linker itself (*i.e.*, it simulates the ld−linux.so interpreter). The third source of slowdown is that the specimen system calls is intercepted by hooking the "INT 0x80" or SYSENTER instructions. In either case, the instruction is intercepted and the simulator processes the system call, either by invoking a real system call on the specimen's behalf, or by adjusting the specimen's state to emulate the system call.

4.2 Usage Scenarios

We describe two sets of usage scenarios to evaluate ROSE's utility and range. First, we describe in detail how a programmer can define an analysis that jumps to an arbitrary offset, then we describe three debugging tactics (which could be turned into tools) that ROSE enables.

4.2.1 Arbitrary Offset

ROSE's power comes at a cost: it is a heavy-weight simulator. Here we show how its ability to begin execution at an arbitrary offset reclaims lost performance. In particular, we show how quickly ROSE can jump to and analyze a function (strcmp, strchr, and updcrc) without inputs in three programs (grep, gzip and vim) as compared to reaching that same function from the program's start under native or simulated execution with concrete inputs.

To begin from an arbitrary offset, ROSE must concretely execute the specimen up to a certain point to resolve dynamic linking. Thus, it parses the ELF file to find the address of main and stops when it is reached. By executing to main, ROSE allows the dynamic linker to run, giving us more information about the executable.

After reaching main, ROSE jumps to updcrc's offset in gzip begins symbolic execution. This function takes a pointer to a buffer and the buffer's size and computes the CRC of the buffer. Since the function has an if statement and a loop, we provide a concrete value for the second argument. This causes the else (the more interesting of the two branches) to be taken and the loop to execute a fixed number of times. We also provide a concrete value for the buffer address, but not the buffer contents.

	grep − glibc strcmp	grep − glibc strchr	gzip updcrc	vim − glibc strcmp
Native	0.003	0.003	0.004	1.09
Simulator	1.809	1.811	3.601	67.00
Offset	1.809	1.808	9.400	6.09

Table 7: Arbitrary offset: time (seconds) to execute the target library call in the given executable; for instance, strcmp from glibc in gzip.

We confirm ROSE's interpretation of functions by comparing the result of the simulation to the result of a concrete run of the same code. Listing 10 shows the salient features of the updcrc function that gzip uses. To evaluate updcrc, we converted crc and crc_32_tab from globals to locals, because ROSE does not currently handle static initializers. To evaluate ROSE's capability to concretely analyze updcrc and provide memory values on the fly, we ran the specimen until it reached main, then allocated memory, initialized it with a string, and analyzed updcrc() to obtain an output value. For instance, when updcrc() is given "hello world!" as input, it produces the CRC "0xbfbe9f4f", the expected value.

Table 7 shows the results of this evaluation. The vim experiment highlights the potential for performance savings that ROSE's ability to begin symbolic execution at an arbitrary offset provides. Guided by Yices, we explored all feasible paths within the function. We also verified that when Yices, our SMT solver, asserted that a path was feasible that its path condition was indeed satisfiable and that when Yices asserted that a path was infeasible that its path condition was indeed unsatisfiable. Yices was able to solve the constraints we fed it in this evaluation.

```
1   struct Demo : public RSIM_Callbacks::InsnCallback {
2       virtual bool operator()(bool enabled, const Args &args) {
3           SgAsmFunctionDeclaration *func =
4               SageInterface::getEnclosingNode<
5                   SgAsmFunctionDeclaration
6               >(args.insn);
7           if (enabled && func!=NULL
8               && 0 == ( func->get_reason() &
9                   SgAsmFunctionDeclaration::FUNC_LEFTOVERS
10              )
11              && !func->get_name().empty())
12                  args.thread->tracing(TRACE_MISC)->mesg(
13                      "in function %s", func->get_name().c_str()
14                  );
15          return enabled;
16      }
17  };
18
19  int main(){
20      ...
21      RSIM_Linux32 sim;
22      int n = sim.configure(argc, argv, envp);
23      sim.install_callback(new Demo);
24      sim.install_callback(new RSIM_Tools::UnhandledInstruction);
25      sim.exec(argc-n, argv+n);
26      SgAsmBlock *top = sim.get_process()->disassemble();
27      ...
28  }
```

Listing 11: Finding the function in which an instruction is executing.

4.2.2 *Debugging with* ROSE

ROSE offers a promising foundation for new debugging tools. Here, we describe three debugging tactics ROSE makes possible. We begin with a description of how to discover in which function a particular instruction is executing. Then we show how ROSE's complete representation of a specimen's storage enables two useful debugging tactics — checking whether a value is present in memory and watching memory accesses.

Which Function Is Mine? When a bug occurs during simulation, it is often convenient to know which function is executing. This example demonstrates how to find the function in which an instruction is executing. In Listing 11, we register a callback that registers the custom analysis function (lines 2–17) to the ROSE simulator on line 23. Since the program might contain instructions that are not handled by ROSE yet, we register a callback to handle those on line 24. After setting up the analysis, we load the executable on line 25 and initiate disassembly of that memory on line 26.

The task of finding an instruction's function has three stages. ROSE provides tool support for each. First, you must disassemble the instructions (with ROSE's Disassembler class) and partition them into basic blocks and functions (with ROSE's Partitioner class). RSIM_Process::disassemble() is a convenience method that does both and caches the result in an RSIM_Process object. Another way to accomplish this task is to use RSIM_Tools::MemoryDisassembler; it triggers RSIM_Process::disassemble() when a specified instruction is hit. If you do not run the disassembler over all of a process' memory, the instruction returned will lack basic block and function information: processing a process' entire memory space is required to build an accurate and complete map of basic blocks and functions. Second, one must acquire a pointer to the instruction. In this example, we write an instruction callback that is invoked for every simulated instruction that takes a SgAsmInstruction pointer as one of its arguments. Finally, once the instruction pointer is obtained, you just walk the AST upward from the instruction to find its SgAsmFunctionDeclaration node.

Is a Particular Value in Memory? To check whether specimen memory matches a known value, one need only attach a MemoryChecker callback to the simulator. The MemoryChecker instruction callback reads from the specified memory area and verifies that the contents of memory at that location match the expected value. If not, a message is printed and the callback is disabled. Here's an example of how to use this callback:

```
uint8_t valid_mem =
    {0x00, 0x00, 0x11, 0x00, 0x00, 0x00, 0x00, 0xef,
     0x67, 0x00, 0x00, 0x00, 0x7f, 0x00, 0x00, 0x03};
MemoryChecker mcheck(
    0x7c402740, sizeof valid_mem, valid_mem, true
);
simulator.install_callback(&mcheck);
```

What Writes a Memory Region? ROSE supports the construction of tools that provide capabilities beyond those of gdb. For instance, gdb can watch memory locations, but ROSE can watch memory regions. This can be particularly useful when trying to discover which code modifies a particular memory region.

The MemoryAccessWatcher callback watches for access to specified memory locations and prints a message to the specified facility when such an access occurs. An access need not change the value of the memory location in order to be reported. This callback is triggered if the bit vector that describes an instruction specifies a memory operation and the watched memory region's protection bits match at least one of the specified req_perms bits.

We ourselves used ROSE's ability to watch a memory region while working on extending ROSE to simulate Windows programs using WINE. At one point a bug made it so that a memory region was incorrectly modified after simulating about a million instructions under WINE. Using MemoryAccessWatcher, we quickly isolated the "POP \%ESP" and discovered that ROSE was incorrectly simulating it. The fix was to swap two lines of code. Without ROSE's memory debugging capabilities, this bug would have been difficult to isolate and fix.

MemoryAccessWatcher can watch arbitrarily large memory regions, since it does not allocate any backing store, but rather just monitors the memory addresses. Also, the addresses to watch do not need to be mapped to the process yet — the memory access watcher can detect access to addresses that would cause a segmentation fault.

An example of how to use a memory watcher to monitor reads and writes to the first page of memory, which are indicative of dereferencing a null pointer, follows:

```
RSIM_Linux32 sim;
unsigned operations = MemoryMap::MM_PROT_READ
  | MemoryMap::MM_PROT_WRITE;
unsigned req_perms =
  MemoryMap::MM_PROT_ANY; //read, write, or execute
RTS_Message mesg(stdout, NULL)
sim.install_callback(new MemoryAccessWatcher(
  0, 4096, operations, req_perms, &mesg
));
```

Here, we have an example of how to use a memory watcher to detect when instructions are executed on the stack, which might indicate a buffer overflow:

```
RSIM_Linux32 sim;
unsigned operations = MemoryMap::MM_PROT_READ;
unsigned req_perms = MemoryMap::MM_PROT_EXEC;
proj_addr_t stack_base = 0xbffeb000;
proj_addr_t stack_size = 0x00015000;
sim.install_callback(new MemoryAccessWatcher(
    stack_base, stack_size, operations, req_perms
));
```

5. RELATED WORK

To our knowledge, ROSE is the only mixed analysis framework that supports the analysis of binary fragments from any offset. Mixed interpretation allows ROSE to simultaneously execute multiple, interacting analyses; ROSE's high-fidelity abstract storage model is also unique and enables ROSE to begin execution starting at arbitrary offsets. The rest of this section surveys related work, which we classify into the following categories.

Virtual Machine ROSE is an extensible virtual machine. There is a wide array of existing virtual machines that support concrete interpretation of binaries. Examples are Bochs [17], Embra [31], QEMU [5], SPIM [28] and PTLsim [32]. However, they all operate on concrete storage, while ROSE operates on abstract storage and supports concrete, symbolic, and abstract interpretations.

Binary Instrumentation Pin [20], Strata [25, 29], DynInst [23] and Valgrind [19] are runtime instrumentation tools that inject instructions into the instruction stream to perform binary analysis. They are tethered to a concrete execution trace although approaches exist that can perturb the program state along the trace to explore more of the program's state space. PinOS [6] can instrument operating systems and unify user/kernel-mode tracers. It is built upon Xen [4] and provides similar functionality as Valgrind. In comparison to these tools, ROSE is a virtual machine where a program is interpreted. Runtime instrumentation approaches do not support abstract storage and thus cannot interpret binaries abstractly or symbolically. Neither can these tools interpret code from arbitrary offsets.

Symbolic Binary Analysis BitBlaze [27], built on top of Valgrind, explores the program execution space around a trace. It is still tethered to the trace and does not support binary analyses of code fragments from arbitrary offsets. Instead, it extracts and changes the program state from a trace to explore the program's additional state space. S^2E [10] is similar to BitBlaze, but adds a more accurate hardware model. S^2E is the first tool that handles all aspects of hardware communication which ROSE delegates to the host OS. S^2E translates between a concrete representation of a program in QEMU and a symbolic representation in KLEE [8], whereas ROSE directly supports symbolic execution of instructions. SAGE [13] is a symbolic execution engine for binaries, developed and used internally at Microsoft. Like BitBlaze, it is also tethered to and operates over concrete traces.

Systematic path exploration techniques for source code, such as DART [14], CUTE [26], SJPF [22] and EXE [9], provide the foundation for the above tools. The basic idea is to synergistically combine concrete and symbolic execution to improve test coverage.

Tools that symbolic execute source code either have to model the services provided by system calls or invoke the system calls directly. For instance, modeling the file system has enabled KLEE to test UNIX utilities without invoking the real filesystem [8]. However, creating models is a labor-intensive and error-prone process and researchers has reported spending several person-years writing a model for the kernel/driver interface of a modern OS [3]. Future work will apply ROSE to the task of simulating a kernel; success here would automatically generate models for external functions including system calls. We currently only generate models for external functions outside the kernel.

Static Binary Analysis As ROSE has a disassembler component, it is also related to disassemblers and PDG-based tools that enable the static analysis of binaries. Notable examples include CodeSurferx86 [2, 24] and IDAPro [15]. However, these tools typically do not have a complete execution model and therefore do not conveniently support semantic interpretation of binaries.

6. CONCLUSION AND FUTURE WORK

In this paper, we have presented ROSE, a pluggable, mixed binary analysis framework. Its novel combination of features allows it to start its analysis in different modes of execution from an arbitrary offset. This capability opens the door to new application domains in the binary analysis arena, such as unit testing of third-party binaries. We believe that ROSE's power, extensibility, and ease of use will enable the design and development of novel and practical binary analysis tools. ROSE is available under the MIT license and can be downloaded from `<anonymized url>`.

References

[1] AMD64: Architecture programmer's manual volume 3: General purpose and system instructions. `http://developer.amd.com/documentation/guides/pages/default.aspx#chipset`.

[2] G. Balakrishnan. *WYSINWYX: what you see is not what you execute*. PhD thesis, Madison, WI, USA, 2007. AAI3278779.

[3] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. *SIGOPS Oper. Syst. Rev.*, 40:73–85, April 2006.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[5] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[6] P. P. Bungale and C.-K. Luk. PinOS: a programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 137–147, New York, NY, USA, 2007. ACM.

[7] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[8] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[10] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. *SIGPLAN Not.*, 46:265–278, March 2011.

[11] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[12] B. Dutertre and L. M. D. Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer Aided Verification*, pages 81–94, 2006.

[13] P. Godefroid, P. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin. Automating software testing using program analysis. *IEEE Softw.*, 25:30–37, September 2008.

[14] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[15] IDAPro: a disassembler. `http://www.idapro.com/`.

[16] Intel IA-32: Architectures software developer's manuals. `http://www.intel.com/products/processor/manuals/`.

[17] K. P. Lawton. Bochs: A portable PC Emulator for Unix/X. *Linux J.*, 1996, September 1996.

[18] LTP: Linux test project. `http://ltp.sourceforge.net/`.

[19] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[20] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 81–92, Washington, DC, USA, 2004. IEEE Computer Society.

[21] POSIX.1-2008: The open group base specifications issue 7. `http://pubs.opengroup.org/onlinepubs/9699919799/`.

[22] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 15–26, New York, NY, USA, 2008. ACM.

[23] G. Ravipati, A. Bernat, B. P. Miller, and J. K. Hollingsworth. Towards the deconstruction of Dyninst. Technical report, 2006. `http://www.dyninst.org/`.

[24] T. Reps, G. Balakrishnan, J. Lim, and T. Teitelbaum. A next-generation platform for analyzing executables. In *In APLAS*, pages 212–229, 2005.

[25] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 36–47, Washington, DC, USA, 2003. IEEE Computer Society.

[26] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. *SIGSOFT Softw. Eng. Notes*, 30:263–272, September 2005.

[27] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, Dec. 2008.

[28] SPIM. `http://pages.cs.wisc.edu/~larus/spim.html`.

[29] Strata. `http://dependability.cs.virginia.edu/info/Strata#Papers`.

[30] Wikipedia: X86 instructions. `en.wikipedia.org/wiki/X86_instruction_listings`.

[31] E. Witchel and M. Rosenblum. Embra: fast and flexible machine simulation. *SIGMETRICS Perform. Eval. Rev.*, 24:68–79, May 1996.

[32] M. T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *in ISPASS '07*, 2007.

Prepared by LLNL under Contract DE-AC52-07NA27344.