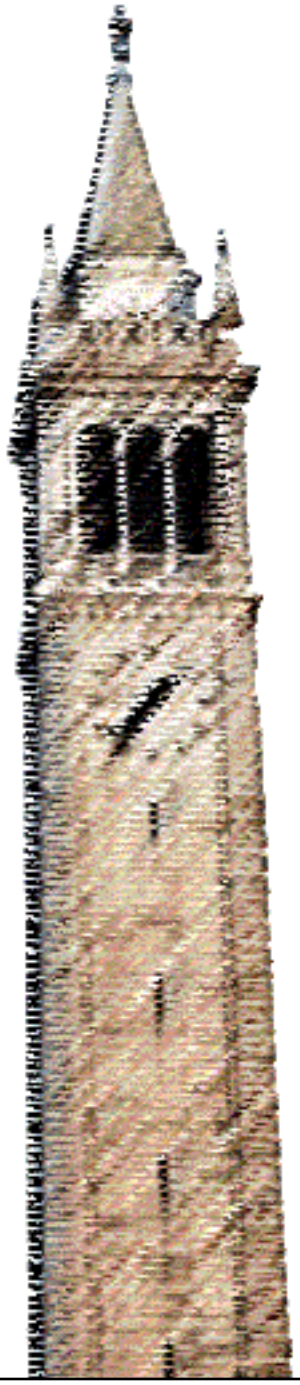
Catchconv: Symbolic execution and run-time type inference for integer conversion errors



David Alexander Molnar
David Wagner

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-23
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-23.html

February 4, 2007

Catchconv: Symbolic execution and run-time type inference for integer conversion errors

David Molnar* and David Wagner**

UC Berkeley and UC Berkeley

Abstract. We propose an approach that combines symbolic execution and run-time type inference from a sample program run to generate test cases, and we apply our approach to signed/unsigned conversion errors in programs. A signed/unsigned conversion error occurs when a program makes control flow decisions about a value based on treating it as a signed integer, but then later converts the value to an unsigned integer in a way that breaks the program's implicit assumptions. Our tool follows the approach of Larson and Austin in using an example input to pick a program path for analysis [21], and we use symbolic execution to attempt synthesis of a program input exhibiting an error [19, 17, 8, 34]. We describe a proof of concept implementation that uses the Valgrind binary analysis framework and the STP decision procedure, and we report on preliminary experiences. Our implementation is available at `http://www.sf.net/projects/catchconv`.

Keywords: Software security, symbolic execution, test generation, decision procedure, dynamic binary analysis

1 Introduction

We present a method of test generation by symbolic execution with run-time type inference from an example program run, describe an application of the method to signed/unsigned integer conversion errors, and report on a prototype implementation. The approach of "symbolic execution" was originally outlined by King [19] and was recently applied by the EXE, DART, and CUTE projects [8, 17, 34] to find bugs in C programs. In symbolic execution, inputs to a program are treated as symbolic values. For each *path*, or sequence of branches taken or not taken in the program, we then collect the *path condition*, which is the set of constraints on the input that must be satisfied for the program to execute that path. Then, to determine if a particular program path leading to an error is feasible or not, we pass the path condition to a decision procedure.

As originally described, symbolic execution focuses on systematic exploration of program paths. The problem with this approach is that it may miss "deep" bugs. For example, consider the problem of testing a Javascript interpreter inside a web browser by loading HTML pages with embedded scripts. Before we can

* dmolnar@eecs.berkeley.edu

** daw@eecs.berkeley.edu

begin testing the interpreter, we must ensure that our test case passes the web browser's HTML parsing code. With systematic path exploration, a tool must "get through" the HTML parsing code before it can start generating Javascript test cases.

Our approach, instead, is to start with an *example input*, collect the path condition followed by the program on that example, and then use this as a starting point for bug-finding. In the web browser example, we might take an existing web page off the web with embedded JavaScript and use this as our example input. We then measure the path condition and attempt to infer "related" path conditions that lead to an error. Here we are inspired by the work of Larson and Austin, who looked for buffer overflow errors by observing length constraints placed on buffers given an example input [21]. We differ, however, in that we look for signed/unsigned conversion errors, which we now describe.

1.1 Signed/Unsigned Conversion Errors

A signed/unsigned conversion error is a sub-class of errors that arise due to the difference between machine arithmetic and arithmetic over the integers. Such an error arises when a program makes control flow decisions based on treating a value as a signed integer, but then treats the same value as an unsigned integer. This can violate implicit assumptions made by the program about the value.

For example, consider the case where a program checks whether a signed integer `i` is less than a constant upper bound, then passes `i` as an argument to `memcpy`. If `i` is negative, then it will pass the bounds check, yet be converted to a large positive integer when passed to `memcpy`. This may in turn overflow a destination buffer. blexim gives an overview of these and other program errors due to machine arithmetic [6].

We recognize that signed/unsigned conversion errors are a narrow class of bug. We focus on this class because it often leads to security issues. Furthermore, as we will see in Section 5, it is possible to use run-time type inference for a simple type system to drive test generation for such errors.

1.2 Problem Statement

The problem we address is the following: given an execution of a program P on an example input x, decide if there exists an input y such that $P(y)$ exhibits a signed/unsigned conversion error. If there exists such an input y, output y.

1.3 Context : Fuzz Testing

Our tool works given an observation of the program P on an example input x. While this would work in principle for any given input x, the context for our work is the practice of *fuzz testing* programs. There are several different types of fuzz testing, including *mutation fuzzing*, where a captured "normal" input is changed before feeding it to the program, and *block-based fuzzing*, where the tester writes

a probabilistic grammar for valid program inputs and the tool generates random productions of that grammar as tests.

The key point in fuzz testing we address is that "near miss" inputs do not typically aid the fuzz tester in finding a bug. Put another way, consider the following over-simplified equation for the expected number of bugs found by a testing method:

$$E[\#\texttt{Bugs}] = \frac{\#\texttt{trial}}{\texttt{time}} \cdot \texttt{time} \cdot \Pr[\texttt{bug per trial}]$$

While this does not by itself capture important information such as the difficulty of exploiting a bug once found, or the cost to fix bugs, it does allow us to think about some testing tradeoffs. Fuzz testers tend to have a high number of trials per time period, but a relatively low chance per trial of exhibiting a bug. Furthermore, in basic fuzz testing hitting a bug does not usually increase the chance that future test cases will exhibit a bug. A recent tool that addresses this is autodafe, which uses `libgdb` to instrument functions such as `strcpy` and determine if test input data affects the arguments to these functions, and then adjusts its testing accordingly [38]. In general, however, directing fuzz testing based on prior test cases is not well understood.

Our work addresses this by using a decision procedure to test for inputs "close" to the example input x which exhibit a bug. If this drives up the probability that a bug is found on a given trial without paying too much in number of trials per time period, we can achieve a win over classical fuzz testing. Put another way, a key question in the background for our work is this: can we combine fuzz testing with symbolic execution to achieve a testing method that does better than either alone?

1.4 Overview

Our test generation approach has four main steps. First, we generate a path condition from observing a program execution on a concrete input x. We use dynamic taint flow analysis to identify execution that depends on untrusted input and so optimize our formula generation. Second, we use a run-time type inference algorithm to identify potential conversion errors. If we discover potential errors, we emit a query formula that is false if and only if an input exists that exhibits an error. Third, we pass the resulting path condition and query to a decision procedure, which then attempts to falsify the query. Finally, if the decision procedure succeeds, we extract a new program input from its answer, then check whether the input in fact triggers an error.

In Section 2 we describe the underlying tools used by our prototype, the Valgrind dynamic binary analysis framework and the STP decision procedure. Then, in Section 3 we outline the generation of STP formulas from Valgrind's intermediate representation. Sections 4 and 5 describe taint flow tracking and run-time type inference. Finally, we report on preliminary experiences with our prototype in Section 6 and cover related work in Section 7.

2 Valgrind and STP

Our prototype depends on two underlying tools, Valgrind and STP [35, 16]. Valgrind is a dynamic binary instrumentation and program analysis framework best known for tracking memory errors, but which is flexible enough to allow implementation of many different program analyses. The current version of our prototype uses Valgrind release 3.2.2 as a starting point, with minor modifications to the Valgrind core.

The second tool is STP, a decision procedure for quantifier-free bitvector arithmetic with arrays. STP accepts a sequence of variable declarations and formulas, followed by a `QUERY` statement. The argument to `QUERY` is a formula ϕ. STP then attempts to decide whether ϕ is valid given the asserted formulas or invalid. If ϕ is invalid, STP provides a counterexample, in the form of an assignment to the variables that satisfies the set of asserted formulas but causes ϕ to be false.

2.1 Valgrind Overview

Valgrind consists of two major components plus an assortment of program analyses, or "tools," built on these components. The first component is the VEX library, which converts blocks of machine code to an intermediate representation. The second component is the Valgrind "core," which handles program loading, interaction with the operating system, and just-in-time translation of the program to VEX intermediate representation, instrumentation, and compilation to machine code.

Valgrind runs as a standard Linux process and acts as a program loader for the program to be analyzed, which is referred to as a *guest*. The Valgrind core then passes each basic block of the guest as it is loaded to the VEX library. The library converts the basic block to an intermediate representation and passes the result to the specified Valgrind tool. The tool then instruments the basic block and passes the result back to the Valgrind core, which uses VEX to compile it to machine code. Valgrind then keeps a cache of compiled instrumented blocks and manages flow of execution between them. In this respect, Valgrind can be thought of as an instrumenting just-in-time compiler.

2.2 Valgrind Intermediate Representation

The VEX library used by Valgrind 3.2.2 converts machine code to a platform-independent intermediate representation. Versions of Valgrind prior to 3.0 used a different intermediate representation called UCode, which differs significantly from the VEX representation used in current versions of Valgrind. We now give a brief overview of the VEX intermediate representation as preparation for describing our tool. Figure 1 shows a subset of the representation in BNF notation, and a complete description is in the `VEX/pub/libvex_ir.h` file in the Valgrind distribution.

The *basic block* is the unit on which a tool operates. A basic block is a sequence of guest program machine code with a single entry point, but which may have multiple exit points. A single machine code operation in a block is translated into one or more `IRStmt` operations. Each `IRStmt` is an operation with side effects, such as storing a value to memory or assigning to a temporary variable. Each `IRStmt` may incorporate one or more `IRExpr` , which are operations with no side effects, such as arithmetic expressions or loads from memory.

Associated with each basic block is a *type environment*. The type environment declares the names of `IRTemp` temporary values in the basic block, and it associates an `IRType` with each `IRTemp`. Examples of `IRType`s include `Ity_I32` and `Ity_I64`, for 32-bit and 64-bit integer values. VEX IR satisfies the single static assignment property; each `IRTemp` is assigned to only once in a single basic block.

Besides `IRTemp` temporaries, VEX IR may refer to guest memory or guest machine state. Memory is accessed through store and load IR operations. Guest machine state is an array of bytes accessed through `PUT` and `GET` operations, which specify an offset into the array for writing and reading respectively. The most common use of the machine state array is to represent reading from and writing to guest machine registers. For example, for an x86 guest, offset 60 represents `eip`, so each machine instruction translated therefore includes a `PUT(60)` statement in its IR representation to set `eip` to its new value.

Finally, VEX supports adding calls to special `IRDirty` statements in a basic block. These are functions with side effects, such as changing memory or printing values to `stdout`. Our tool makes extensive use of such functions to update metadata about the program's execution and to emit formulas for STP.

The basic mode of operation for a Valgrind tool is as a VEX-to-VEX transformation. First, the tool registers a callback with the Valgrind core. Each time a new basic block of machine code is ready for JIT'ing, Valgrind does a preliminary conversion to IR and then calls the tool. The tool inspects the resulting sequence of IR statements, then updates its metadata, adds or deletes IR statements from the basic block, and finally returns. The Valgrind core then uses the VEX library to sanity check the basic block and compile it to machine code before finally executing the result.

2.3 Concurrency and Syscalls

Concurrency and syscalls require special handling from the Valgrind core, because the core must retain control of program execution. In the case of concurrency, the core intercepts signals and passes notifications to the tool. The core also emulates `fork` within the host process.

Many syscalls, however, cannot be fully emulated by the core. Instead, Valgrind comes with a library of syscall annotations for Linux. Each annotation specifies whether the syscall writes to memory or guest state, and if so, at which addresses. Tools can then register callbacks that are invoked after each such side effect. While this approach was originally developed for Valgrind's Memcheck memory checking tool, we found that this abstraction was well-suited for our

```
IRStmt :==  NoOp
| IMark of Addr64 * Int
| AbiHint of IRExpr * Int
| Put of Int * IRExpr
| PutI of IRArray * IRExpr * Int * IRExpr
| Tmp of IRTemp * IRExpr
| Store of IREndness * IRExpr * IRExpr
| Dirty of IRDirty
| MFence
| Exit of IRExpr * IRJumpKind * IRConst

IRExpr :== Binder of Int
| Get of Int * IRType
| GetI of IRTemp * IRArray * IRExpr * Int
| Tmp of IRTemp
| Qop of IROp * IRExpr * IRExpr * IRExpr * IRExpr
| Triop of IROp * IRExpr * IRExpr * IRExpr * IRExpr
| Binop of IROp * IRExpr * IRExpr
| Unop of IROp * IRExpr
| Load of IREndness * IRType * IRExpr
| Const of IRConst
| CCall of IRCallee * IRType * IRExprVec
| MuxOX of IRExpr * IRExpr * IRExpr

IRExprVec :== IRExpr | IRExprVec
IREndness :== LittleEndian | BigEndian
IRArray :== Int * IRType * Int
IRTemp :== UInt
IRConst :== Bool | UChar | UShort | UInt | ULong | Double

IRJumpKind :== Ijk_Boring | Ijk_Call | Ijk_Ret
| Ijk_ClientReq | Ijk_Yield
| Ijk_EmWarn | Ijk_NoDecode | Ijk_MapFail | Ijk_TInval
| Ijk_NoRedir | Ijk_Trap | Ijk_Sys_syscall | Ijk_Sys_int32
| Ijk_Sys_int128 | Ijk_Sys_sysenter

IRType :== Ity_INVALID | Ity_I1 | Ity_I8 | Ity_I16 | Ity_I32
| Ity_I64 | Ity_I128 | Ity_F32 | Ity_F64 | Ity_V128
```

Fig. 1. A subset of VEX intermediate representation in BNF notation. Definitions for `IRDirty`, `IROp`, and `IRCallee`, in particular, are not shown. The header file `VEX/pub/libvex_ir.h` contains an annotated description.

```
0x8048102:  popl %esi

      ------ IMark(0x8048102, 1) ------
      PUT(60) = 0x8048102:I32
      t4 = GET:I32(16)
      t3 = LDle:I32(t4)
      PUT(16) = Add32(t4,0x4:I32)
      PUT(24) = t3
```

Fig. 2. Translation of x86 `popl` instruction at address 8048102 to VEX intermediate representation. The instruction is rendered as five `IRStmt` operations: a `PUT` to write the instruction pointer to guest offset 60, a read from guest offset 16, a load from memory into `IRTemp t3`, and then storing to offsets 16 and 24. The suffix `I32` indicates that the value is a 32-bit integer. The header `VEX/pub/libvex_guest_x86.h` reveals that offset 16 corresponds to the register `esp`, offset 24 is `esi`, and offset 60 is `eip`.

purposes as well. Specifically, these notifications allow us to emit appropriate "fixup" formulas for exactly the affected locations after each syscall.

2.4 STP Overview

STP is a decision procedure for bitvector arithmetic with arrays. A *bitvector* is a sequence of Boolean variables. STP works with formulas over such variables which include a variety of predicates, including arithmetic and comparison. Given a sequence of formulas asserted to be true and a *query formula*, STP attempts to determine whether the query formula is *valid*, i.e. whether it is necessarily true given the asserted formulas. If the query is *invalid*, then STP provides a counterexample, in the form of an assignment to the variables of the asserted formulas that makes the query false. The algorithms in STP have been tailored for the needs of program analysis tools; STP is the decision procedure used by the EXE, Replayer, and Sweeper tools, and test cases from these tools as well as from Catchconv have influenced its development [8, 37, 30].

2.5 STP Presentation Language

We now give a brief overview of the subset of the STP input language used by the Catchconv tool. While STP can be linked as a library with a tool, we chose to emit formulas as ASCII text and pass them to STP for ease of debugging. In particular, we can quickly troubleshoot syntax errors by editing input files and re-running STP. For a full overview, see the STP web site [16].

The STP input language is similar to that used by CVC Lite, but with some additions [3]. STP supports variable declarations with type `BITVECTOR(X)`, meaning an array of `X` boolean variables, and of type `BOOLEAN`. Declarations may be mixed with `ASSERT` statements; each `ASSERT` takes as an argument a formula involving variables previously declared.

Formulas in STP can use a variety of arithmetic predicates and comparison operations. Of special concern for us is that STP offers native support for both signed and unsigned bitvector comparisons. For example, `BVLT` is an unsigned less-than comparison predicate, but `BVSLT` is a signed less-than comparison.

STP also supports bitvector arrays that are themselves indexed by bitvectors. To use such arrays, we first declare a new variable of type `ARRAY BITVECTOR(X) OF BITVECTOR(Y)`, where `X` is the size of the array address and each entry is a bitvector with `Y` bits. STP allows non-constant indices for arrays, such as bitvector variables of the appropriate length or sub-ranges of longer bitvector variables. This makes it straightforward to represent loading a value from a symbolic memory address in STP; we simply read from an address represented as a variable of type bitvector.

Writes to an array directly are not supported by STP. Instead, STP offers a special syntax for array updates. An update takes the form of a declaration of a new array, together with a statement that the new array is identical to an old array except for a specified entry. For example, the following declarations

```
A : ARRAY BITVECTOR(4) OF BITVECTOR(32);
B : ARRAY BITVECTOR(4) OF BITVECTOR(32) = A WITH [0hex5] := 0hex00000001;
```

state that the array `B` is an update of the array `A` with a new value for entry `0hex5`. We will refer to these types of declarations in the paper as *update constraints*.

```
i : BITVECTOR(32);
j : BITVECTOR(32);

ASSERT(BVSLT(i,0hex0000000a));
ASSERT(i = j);
QUERY(BVLT(j,0hex00000032));
```

Fig. 3. A tiny example of an STP input file. This file declares two 32-bit bitvector variables `i` and `j`, then asserts that `i` is less than 10 as a signed integer. Finally, it asserts that `i` equals `j` and emits a `QUERY` asking whether `j` as an unsigned integer is always less than 50.

Finally, STP inputs may contain at most one `QUERY` statement, which takes a formula as its argument. STP then reports whether the `QUERY` is valid or invalid given the formulas previously `ASSERT`ed.. Figure 3 shows a small example STP input, while Figure 4 shows the output of STP.

3 Formula Generation From VEX IR

The first step of our approach is the generation of formulas that capture the execution of a program P on an example input x. Our prototype implements

```
[dmolnar@glimmung catchconv] ~/solvers/smt/stp/stp-01-17-07 -p quicktest
Invalid.
ASSERT( i  = 0hex8000004C  );
ASSERT( j  = 0hex8000004C  );
```

Fig. 4. Output from running STP on Figure 3. The `-p` flag asks STP to print an assignment to the bitvector variables which makes the `QUERY` false. The `QUERY` is invalid because `0hex80000004C` is negative, and so less than 10 when interpreted as a signed integer, but greater than 50 when treated as unsigned.

formula generation as a map from VEX intermediate representation to STP formulas. For each VEX statement, we define a corresponding STP formula that captures the semantics of the IR statement. We then instrument basic blocks of the program with helper functions that print `ASSERT` statements with these STP formulas to `stdout`.

Figure 5 shows an example translation. Here, a x86 `popl` instruction is translated to VEX IR, and then to a set of STP formulas. This example shows how guest memory and guest state are modelled by STP arrays.

3.1 Memory Regions

The naive approach to modeling memory of a 32-bit machine with STP is to declare an array indexed by 32-bit bitvector values. Each load expression translates into a read from the corresponding array entry. Each store then translates into an update of the corresponding array entry. The main advantage of this approach is that it allows for exact pointer arithmetic and bit-level precision modeling of symbolic memory.

Unfortunately, this naive approach also leads to performance problems. Despite the fact that the array is extremely sparse, we have found that using a single array for all of memory leads to STP hitting the 3 GB per-process memory limit on our 32-bit test machine. Therefore, we use an optimization found in the EXE tool, namely that of dividing memory into different *regions* [8]. Each region is modelled by a separate bit-vector array. We also keep track of a base address for each region and a total size for the region. The main benefit of memory regions is that they free STP from having to reason about aliasing between reads and writes in different memory regions. The main downside is that use of regions can lead to a loss of precision.

Loads from and stores to memory then pass through three steps in their conversion to STP. First, we use the concrete value of the memory address to select the appropriate memory region. Second, we declare a new *offset variable* of the appropriate size to be an offset into memory region. Finally, we `ASSERT` formulas that set the offset variable to the value of the original address minus the base address of the region. Figure 6 shows an example of the result.

We can also emit `QUERY` statements that ask whether the resulting offset is always within the bounds of the memory region (not shown). If the `QUERY` is

```
        0x8048102:  popl %esi

------ IMark(0x8048102, 1) ------
PUT(60) = 0x8048102:I32
t4 = GET:I32(16)
t3 = LDle:I32(t4)
t25 = Add32(t4,0x4:I32)
PUT(24) = t3

MemState0p22034th1 : ARRAY BITVECTOR(32) OF BITVECTOR(32);
MaSt0p22034th1 : ARRAY BITVECTOR(12) OF BITVECTOR(32);
[...]
MaSt2p22034th1 : ARRAY BITVECTOR(12) OF BITVECTOR(32) =
MaSt1p22034th1 WITH [0hex03C] := 0hex08048102;

ASSERT(CV0e1t4p22034th1 = MaSt2p22034th1[0hex010]);
ASSERT(CV0e1t3p22034th1 = MemState0p22034th1[CV0e1t4p22034th1]);
ASSERT(CV0e1t25p22034th1 = BVPLUS(32,CV0e1t4p22034th1,0hex00000004));

MaSt3p22034th1 : ARRAY BITVECTOR(12) OF BITVECTOR(32) =
MaSt2p22034th1 WITH [0hex018] := CV0e1t3p22034th1;
```

Fig. 5. Result of translation from VEX IR to STP formulas for `popl` instruction. Each `IRTemp` is given a name that encodes the number of the basic block as translated (e.g. `CV0`), the number of basic blocks executed so far (e.g. `e1`), the name of the `IRTemp` , the process PID, and the current thread ID. The `MemState` and `MaSt` arrays model guest memory and guest state, respectively.

```
MemRegion0p22034th1baseBEFFEAE8 : ARRAY BITVECTOR(16) OF BITVECTOR(32);
[...]
MaSt2p22034th1 : ARRAY BITVECTOR(12) OF BITVECTOR(32) =
MaSt1p22034th1 WITH [0hex03C] := 0hex08048102;

ASSERT(CV0e1t4p22034th1 = MaSt2p22034th1[0hex010]);

% LoadAddr BEFFF790 Type I32 Value 00000002 RegionBase BEFFEAE8
CV0e1t4p22034th1OFFSET : BITVECTOR(16);
ASSERT(CV0e1t4p22034th1OFFSET = BVSUB(32,CV0e1t4p22034th1,0hexbeffeae8));
ASSERT(CV0e1t3p22034th1 =
MemRegion0p22034th1baseBEFFEAE8[CV0e1t4p22034th1OFFSET]);

ASSERT( CV0e1t25p22034th1 = BVPLUS(32,CV0e1t4p22034th1,0hex00000004));

MaSt3p22034th1 : ARRAY BITVECTOR(12) OF BITVECTOR(32) =
MaSt2p22034th1 WITH [0hex018] := CV0e1t3p22034th1;
```

Fig. 6. Translation of `popl` instruction to STP formulas with memory regions. The 32-bit load address in variable `CV0e1t4p22034th1` is translated to a 16-bit offset by subtracting the base address of the memory region. The region used is chosen by the concrete value of the load address at execution time. We could optionally emit a `QUERY` to check that the offset is always smaller than the size of the memory region.

falsified, then there is an input that causes an out-of-region memory access. At the least, this indicates imprecision in our modeling, and at worst may indicate a program error such as an out-of-bounds memory access.

Our current implementation defines regions in two ways. First, we intercept calls to `malloc` and related functions. Second, the tool takes as an auxiliary input a user-defined list of memory region base addresses and corresponding sizes. We have also written a small script that graphs memory addresses accessed given output by our tool. We have used this graph successfully to craft memory regions "by eye" for small test cases.

3.2 Bug Oracles

The Valgrind framework allows us to conduct run-time checks to reveal whether a specific concrete input triggers a bug. By using these checks as bug oracles, we can ensure that an input in fact causes a bug before reporting that input to the user. While a wide variety of checks are possible, our prototype focuses on two. First, we intercept `malloc()` and related functions and check whether the high bit of the number of bytes to allocate is set, i.e. whether the argument is negative. Second, we check for segmentation faults in the guest program.

3.3 Recording the Path Condition

For each conditional exit statement, our per-basic-block instrumentation declares a boolean variable named `JUMPCONDXeYcZ`, where X is the number of the basic block translated, Y is the number of basic blocks executed so far, and Z is the number of previous `JUMPCOND` variables declared. We need both Y and Z because a single basic block may have multiple conditional exit statements. Our tool, however, cannot tell at instrumentation time whether the exit will be taken or not taken, because we do not yet know the concrete value of the guard expression for the exit statement. Therefore, we need to add instrumentation that records at execution time whether the exit is taken or not taken so that we can emit the correct path condition later.

To do so, we borrow an idea from the Valgrind `lackey` tool. At Valgrind startup, we initialize a hash table to hold the path condition. Each node of the hash table holds the name of a `JUMPCOND` variable and the status "Taken" or "Not Taken." We insert a helper function before the exit `IRStmt` that inserts the corresponding `JUMPCOND` with the status "Taken." We then insert a helper function following the exit that changes the status to "Not Taken."

Assuming that execution is straight-line within a single basic block, the second helper is executed if and only if the exit is not taken. This gives us a record of which exit statements were taken and which not taken during execution. We use this to assert the appropriate path condition after guest execution has ended.

3.4 Future Improvements

Our prototype emits formulas for many VEX IR statements, but not all. In particular, we do not yet handle multiplexor expressions, such as `MuxOX`, we do not translate guest state indirect `PUTI` or `GETI` operations, and we do not accurately model floating-point arithmetic. There are also helper expressions used by VEX to model certain machine updates, such as calculation of condition flags, that are not transformed to STP. Instead, we capture the concrete value of these calls and emit a constant assignment. This leads to a loss of precision in modeling guest execution. To fix these issues, we need to write helper functions that emit the appropriate STP formulas when these IR statements are executed.

Modeling memory as an array `BITVECTOR(32) OF BITVECTOR(32)` raises at least two issues. First, non-aligned memory loads may not see the effects of preceding stores, because each address points to a separate `BITVECTOR(32)` value. Second, if all loads are guaranteed to be word-aligned, then it is not necessary to have a full `BITVECTOR(32)` for each address. As such, our modeling choice is imprecise. Similar issues apply to memory regions. Our original rationale for this choice was ease of implementation: with this modeling, temporary variables can be directly used as addresses, and can be destinations and sources for memory assignments

With respect to the first issue, one approach would be to declare memory as an array of `BITVECTOR(8)` instead of an array of `BITVECTOR(32)`. Then each VEX store would be rendered as an update to the appropriate entries of the

array; for example, a 32-bit store would emit four update constraints, one for each byte. This has the advantage of being straightforward to implement, at the cost of quadrupling the number of array update constraints and introducing a significant number of bitvector extraction operations. A second approach would be to detect loads that are not word-aligned and compile them into assignments from sub-ranges of the appropriate `BITVECTOR(32)` variables. For constant index loads, this is straightforward, but variable indices seem to require embedding a case analysis into the generated formulas. This second approach would also allow us to shorten the size of array address variables.

We faced a similar issue when deciding whether to model guest machine state as an array of `BITVECTOR(32)` or `BITVECTOR(8)`. In particular, x86 registers can be subregisters of other registers : for example, `ch` is a subregister of `ecx`. We discovered that this led to inconsistent formulas following executions that updated `ecx` and then read from `ch`. Preliminary test cases, however, found a significant slowdown for modeling machine state as a vector of `BITVECTOR(8)`. Instead, we used the second approach, buoyed by the fact that all accesses to machine state in our test cases are for constant offsets, and as such detecting non-word-aligned reads to the array is easy. We further discovered that for our test cases, the number of "non-aligned" guest state accesses is small compared to the total number of accesses, so the fact that the second approach incurs no overhead on aligned accesses is a win.

For memory regions, a clear improvement would be automatic generation of memory regions corresponding to the stack or individual stack frames. Together with the memory regions we already generate by intercepting `malloc`, this would significantly reduce the need for hand-generated memory regions. Ideally, this improvement would mean that hand-generated memory regions become unnecessary. Implementing this requires a method for determining the size of the program stack and detecting stack switches by the program. Fortunately, Valgrind already provides infrastructure for reporting changes in the stack pointer, and the Memcheck tool has some example heuristics for detecting stack switches.

Our tool does not yet automatically translate counterexamples emitted by STP into new program inputs. To fix this, we need to write scripts to parse the output of STP, identify variables in the counterexample corresponding to tainted inputs, and then create new program inputs. A stopgap measure is to look at the assignments in the counterexample to memory locations marked as symbolic, and then use those assignments to create new program inputs.

The performance of guest programs could be improved. Currently, our tool translates VEX IR to STP formulas on a statement-by-statement basis. While easiest to implement, we add at least one new `IRDirty` helper for each `IRStmt` in the original basic block. This leads to an extreme slowdown for the guest program. In principle, we could instrument the guest to record only the identity of the basic block executed and memory operations, then perform the translation to STP formulas after the fact. This is important if we are interested in example inputs generated by interacting with humans. This may also be useful if we have

a way to "pre-filter" inputs before starting the translation to STP, as then we can raise the number of inputs tested per time unit.

Finally, we could output formulas in SMT-LIB format to facilitate comparison of different decision procedures. While there is a CVC2SMT tool that converts the STP presentation language to SMT-LIB format, this tool segfaults on the test cases we have generated. This would also allow us to contribute test cases to the SMT-LIB competition.

4 Optimization : Taint Tracking

We have described how to translate program execution to a set of STP formulas. In practice, however, only a fraction of the program's execution depends on untrusted values. Our observation is that if we can identify locations that are independent of untrusted values, then we can simplify the STP formulas generated by the tool.

To do so, we implement a basic dynamic taint flow analysis. In such an analysis, we start by marking a set of memory locations as taint *sources*. During execution, we propagate taint as follows: program locations that read from tainted program locations become tainted, while locations that are overwritten with untainted values become untainted.

4.1 Valgrind Implementation

At Valgrind startup, we initialize a hash table that maps *program locations* to "tainted" or "untainted." A program location is a memory address, a guest state offset, or an `IRTemp`. Our tool then instruments each side-effect generating VEX IR statement with a helper function that checks whether the statement's arguments are tainted or not tainted. If they are tainted, the helper function marks the destination program location as tainted. If all arguments are not tainted, then the helper function marks the destination as not tainted. We then expose an interface to the hash table that allows our formula generation to check at any point during execution whether a program location is tainted or not tainted.

We note that Valgrind's Memcheck tool does a similar kind of taint flow to determine "definedness" of memory addresses, i.e. whether an address is a valid target for memory accesses [35]. While we were inspired by Memcheck, we found it easier to simply implement our own taint flow analysis from scratch rather than use Memcheck directly. One issue we found was that Memcheck keeps track of taint for `IRTemp` values in special "shadow temps"; while this significantly speeds execution of the guest program, it complicates determining whether a particular `IRTemp` is tainted or not tainted at formula generation time.

Figure 7 shows an example of how our current prototype uses taint information to simplify formula generation. Just as in Figure 5, these represent an x86 `popl` instruction. In this case, however, neither the target of the `GET` instruction nor the memory address loaded are tainted.

```
MaSt2p22034th1 : ARRAY BITVECTOR(12) OF BITVECTOR(32) =
MaSt1p22034th1 WITH [0hex03C] := 0hex08048102;

% EmitTmpAssignConcrete. lhstype : I32 value : BEFFF790
ASSERT(CV0e1t4p22034th1 = 0hexBEFFF790);

% LoadAddr BEFFF790 Type I32 Value 00000002 RegionBase BEFFEAE8
ASSERT(CV0e1t3p22034th1 = 0hex00000002);
ASSERT( CV0e1t25p22034th1 = BVPLUS(32,CV0e1t4p22034th1,0hex00000004));

MaSt3p22034th1 : ARRAY BITVECTOR(12) OF BITVECTOR(32) =
MaSt2p22034th1 WITH [0hex018] := CV0e1t3p22034th1;
```

Fig. 7. Translation of `popl` instruction on untainted inputs. Notice how reads from arrays have been replaced by constant values.

We use taint flow information to suppress the generation of array update constraints when a memory value or register value does not depend on untrusted input. In these cases, it is safe to assign the concrete value to the appropriate destination. Figure 7 shows an example of the resulting formulas.

4.2 Future Improvements

First, we could take advantage of the fact that each basic block is JIT'ed as a unit. Therefore, instead of adding a helper function after each IR statement, we could add a call to a single function that propagates taint for the basic block as a whole[1]. The goal of this optimization would be to improve the speed of the guest program's execution. We could also look again at leveraging the infrastructure used by Memcheck for tracking definedness.

Second, we could be more aggressive in using taint information to prune the set of formulas. For example, we could simply not emit formulas for values which both arise from untainted reads and are never arguments to operations on tainted data. Currently, our prototype does emit these expects STP to determine which can be safely dropped. A rough check suggests that we may be able to cut the size of our test cases by an order of magnitude.

Finally, the current handling of system calls is primitive. We only treat the result of `read` system calls as tainted, and we do not take the arguments to system calls into account when deciding whether to mark memory locations as tainted or not. For example, we might want to specify that only reads from certain files should be marked as tainted.

[1] We thank Daniel Wilkerson for pointing out this optimization.

5 Optimization : Type Inference

Given the path condition, we can emit a `QUERY` formula involving any memory location, register, or IR temporary value we like. To find signed/unsigned conversion errors, however, we need to emit the correct `QUERY` for the correct location. We now describe a run-time type inference approach for identifying program locations of interest, and our tool's current implementation of that approach.

5.1 Signed/Unsigned and a Four-Point Lattice

Our basic approach is to consider four types for integer-valued program locations : "Unknown," "Signed," "Unsigned," or "Contradictory." Here "Signed" means that the location has been used in a way "consistent with" the value being a signed integer, while "Unsigned" means the location has been used in a way "consistent with" being an unsigned integer. By "consistent with," we mean that the location has been the source of an argument to a signed or unsigned comparison.

These types form a simple four-point lattice, with "Unknown" corresponding to $\top$ and "Contradictory" corresponding to $\bot$. Our goal is to determine the set of program locations during a concrete execution that have type $\bot$. These program locations are candidates for signed/unsigned conversion errors.

One of the attractive features of the four-point lattice is that it is easy to solve the resulting type constraints "on the fly." We associate each program location with a type variable. Then, when we update a type variable based on observing the location used as signed or unsigned, we can simply set the variable to the meet of its old type and the new type. It is not necessary to keep a set of constraints associated with each variable for later solving.

5.2 Valgrind Implementation

At Valgrind startup, we initialize two hash tables. The first maps program locations to type variables. The second maps type variables to elements of the four-point lattice. We then expose interfaces to query and update both hash tables.

We then implement run-time type inference in two parts. The first part adds instrumentation to manage the mapping from program locations to type variables. If we copy a value from one location to another, we set both locations to map to the the same type variable. If we perform an operation on a location, or we read from a location not previously seen, we declare a new type variable and update the mapping accordingly.

The second part adds instrumentation to update the type value of each type variable. For a signed comparison, we set the type variables of each argument to the meet of their current types and "Signed ." We perform a similar update for arguments to an unsigned comparison. For binary operations, we leave the types of the arguments alone, but we set the type of the destination to "Unknown."

```
int main (int argc, char** argv)
{
        int i = atol(argv[1]);
        unsigned int j = 0;

        if (i < 10)
        {
                j = i;
                if ( j > 50)
                {
                 printf("Surprise! \n");
                 return 1;
                }
        }
return 0;
}
```

Fig. 8. Simple test case program for dynamic type inference and query generation. If `argv[1]` is positive and greater than 10, then only the signed comparison `i < 10` is executed. In contrast, if `argv[1]` is between 0 and 10, then the signed comparison `i < 10` and unsigned comparison `j > 50` emit contradictory type constraints for the same type variable. The resulting type of $\perp$ then causes Catchconv to emit a `QUERY` .

Finally, if a type variable is set to $\perp$, our instrumentation emits a `QUERY` for the program location involved in the most recent type update. The `QUERY` asks the solver to determine if the program location is always a positive signed integer value. If the `QUERY` is false, therefore, there is a program input that causes the program location to contain a negative value; this negative value will cause a divergence between signed and unsigned comparisons.

Our type inference is not guaranteed to be sound or complete. Because we use a decision procedure to generate new program inputs from invalid `QUERY` statements, however, we can test these inputs on the program and filter our false positives. The key tradeoff then is between the cost of false negatives due to faulty type inference and the cost to answer valid `QUERY` statements.

5.3 Future Improvements

Currently, our type inference sets the type for the destination of all binary operations to "Unknown." A simple improvement would be to make this configurable on a per-operation basis. The choice of which rules to use for binary operations then becomes an empirical question that can be investigated on test programs of interest. We could also emit `QUERY` statements for all program locations associated with a particular type variable, instead of only the location most proximate to the assignment to $\perp$.

Second, we realized during the implementation of our tool that a cleaner way to implement type variable propagation would be to use a union-find data struc-

ture. In effect, we would keep track of equivalence classes of program locations with respect to typing information, and associate a single type variable with each equivalence class. Besides being more elegant, such an implementation may be faster than our current hash table mapping each location to a type variable.

Third, our tool currently emits the entire path condition in addition to `QUERY` statements. This includes assertions for `JUMPCOND` variables that occur logically following the `QUERY` of interest. As a result, the current formula outputs need to be edited by hand to remove these anachronistic `JUMPCOND` values, or else the formulas may unintentionally preclude possible inputs that could cause the `QUERY` to be false. Fortunately, both the location named in the `QUERY` and the `JUMPCOND` encode the number of total basic blocks executed in their name, so filtering the path condition is straightforward.

Finally, the four-point lattice is only one possible type system. We could extend our approach to test generation for other type systems. In principle, any type system that admits run-time inference and an STP query for inputs that cause a type violation could be used. The key challenge here is identifying type systems that capture security properties, admit efficient run-time inference, and which also lead to queries efficiently solvable by STP.

6 Tool Status

```
------ IMark(0x8048102, 1) ------
PUT(60) = 0x8048102:I32
DIRTY 1:I1 ::: ogEmitPutConstConstraints{0x380073d9}(0x3C:I32,0x0:I32)
DIRTY 1:I1 ::: cgEmitPutConstStmt{0x3800222e}(0x3C:I32,0x8048102:I32,0x11003:I32)
t4 = GET:I32(16)
DIRTY 1:I1 ::: ogTmp2GetAliasHelper{0x38006a6c}(0x4:I32,0x10:I32,0x0:I32)
DIRTY 1:I1 ::: isFlowMapByKey{0x38007925}(0x10:I32,0x10000:I32)
DIRTY 1:I1 ::: cgEmitTmpAssignConcrete{0x38004d99}(0x4:I32,0x0:I32,0x11003:I32,t4)
DIRTY 1:I1 ::: cgEmitTmpGetConstraints{0x380048ce}(0x10:I32,0x4:I32,0x0:I32,0x11003:I32)
```

Fig. 9. Part of a basic block after instrumentation by Catchconv. Each `IRDirty` statement will be turned into a call to the named function by the VEX library and compiled to machine code. In this example, `ogEmitPutConstConstraints` and `ogTmp2GetAliasHelper` handle type inference, `cgEmitPutConstStmt`, `cgEmitTmpAssignConcrete`, and `cgEmitTmpGetConstraints` emit STP formulas, and `isFlowMapByKey` checks for taint and propagates if necessary.

Our tool currently comprises 6223 lines of code, as measured by SLOCCount [39], developed over roughly five months of work. The majority of this code consists of the Catchconv tool implemented on top of Valgrind 3.2.2. Figure 9 shows a fragment of a basic block post Cachconv instrumentation. The tool is available on Sourceforge at `www.sf.net/projects/catchconv` under the GNU General Public License version 2.

We have developed toy examples to test type inference, such as the one shown in Figure 8. Our tool successfully generates `QUERY` statements from type inference in these test cases. We noticed that the tool emits `QUERY` statements for several pieces of code that are dynamically linked with our test cases, but we have not yet followed up to attempt test generation for these queries. We can also generate path conditions for some small but real programs such as `gzip`, the `thumbnail` program in the libTIFF 3.8.3 distribution, `cat`, and `ls`.

We have been able to generate new candidate inputs exhibiting a bug from a `QUERY` for the test case shown in Figure 8, but not yet for larger programs. For this test case, STP on our test machine `koschei.cs.berkeley.edu` reports the query invalid and generates a counterexample in 6260.2 seconds, using 2238.38 MB of memory on 105947 non-comment lines of STP input. Our parsing of counterexamples to yield new test inputs is currently primitive, but simply looking at all assignments to bitvector variables associated with symbolic memory addresses yields 40 distinct new trial inputs, of which 4 in fact trigger the "bug" in the test case.

The main issue with our current prototype is that the resulting test cases are too large. For example, the output from Catchconv from observing `gzip` or `ls` on small inputs are in excess of 900 megabytes, while the outputs from observing `thumbnail` are 300 to 400 megabytes in size. Even the test case shown in Figure 8, statically linked, generates a constraint set 6.6 megabytes in size.

This has led to two problems when attempting to solve the resulting constraints. The first problem is that STP may quickly hit the 3 gigabyte per-process memory limit during initial processing of the test case on our machine, `koschei.cs.berkeley.edu`, running Red Hat Enterprise Linux 4 with 4 gigabytes total RAM. Recently, however, new versions of STP have been able to successfully solve many of these test cases, including all of the `thumbnail` test cases, without hitting the memory limit.

The second problem we have observed is that on some test cases, STP enters a tight loop of generating CNF formulas and then calling the MiniSat solver. By itself, this is not a problem. The issue is that each iteration of the loop grows the memory footprint of STP by a small but significant amount. After several hours, STP hits the 3 G memory limit and errors out. We are currently investigating the limits of this phenomenon. An alternative approach would be to move to a 64-bit machine, but we prefer to focus on algorithmic improvements first.

Both problems appear to stem from the fact that our current queries include formulas from translation of *all* code executed during the program run, not just the code of interest. For example, with our current taint tracking settings, glibc code that reads from disk as part of dynamic linking will become tainted and emit formulas with array updates, even if that code has little to do with the program of interest. We plan to pursue more aggressive methods of pruning formulas before passing them to STP.

A second issue is the program slowdown due to the tool. We have made little effort to optimize the running time of programs inspected by Catchconv. For testing and development purposes, the resulting slowdown is annoying but

livable. For improving fuzz testing, however, this slowdown directly affects the expected number of bugs we will find in a given period of time. One simple improvement here is to delay printing STP formulas until the end of program execution; this would remove the overhead from calling the Valgrind `printf` function after most IR instructions. As mentioned, another approach would be to reduce the amount of guest instrumentation and do more conversion to STP formulas after the fact.

Our tool does not currently work correctly with programs that use `fork`. While the tool does emit output in this case, the resulting formulas do not pass STP syntax checking. The issue here seems to be that some output is not printed to `stdout` correctly during the Valgrind core's emulation of `fork`.

Finally, we have discovered a few programs where Catchconv causes a translation error in the VEX library. Examples include `bzip2` and the `cjpeg` program in the libjpeg distribution. We have not yet investigated the cause of these errors.

6.1 Test Cases

Over the course of developing our prototype implementation, we have created over twenty test cases for the STP solver. These test cases reflect different decisions regarding constraint generation, such as whether to use memory regions or not. The running time of STP on these test cases has helped us drive decisions regarding formula generation, such as whether or not to implement memory regions. Our test cases have also been helpful in communicating the needs of our tool to the authors of STP. The resulting improvements in STP have been dramatic: one new version of STP cut running times on our test cases from in excess of one hour to less than 3 minutes.

These test cases can be found in the `catchconv-cases` project in the Sourceforge `catchconv` repository. Each test case is stored in `.tar.bz2` form, and includes the source file for the program under analysis with a short description of the test case. Also included are scripts to extract the test cases and run STP on the results.

We have also created scripts that compute statistics related to the use of arrays in each test case. For example, we compute the number of read and update constraints, and the number of such constraints with non-constant indices. This is important because a read or update with a non-constant index involves reasoning about aliasing. The `doreport` script in the `catchconv-cases` project automatically computes and displays these statistics.

7 Related Work

King outlined the basic idea of symbolic execution for program testing and gave a simple implementation called EFFIGY [19]. More recently, several projects have focused on symbolic execution for programs written in C. The EXE project at Stanford uses the CIL front end to parse C code and emit formulas for the same STP decision procedure that we use [8]. Also closely related are DART

and CUTE [17, 34], which look at integrating random testing with symbolic execution. These projects attempt to explore the space of program executions looking for errors; EXE in particular uses integer overflows as a heuristic to drive its search.

The main difference between our work and these projects is that we do not focus on systematic exploration of program executions. Instead, we attempt to generate a bug exhibiting input given an observation of a program run on a specific example input. The most directly related project to ours is the work of Larson and Austin, who generated array size constraints from observation of a program run on a concrete input to look for possible buffer overflow errors [21]. We look for a different class of errors, and we attempt to actually synthesize an input exhibiting the bug. A secondary difference is that these projects focus on source-level analysis, while our use of Valgrind means that we can analyze Linux binaries.

Decision procedures have also been used to reduce the number of false positives from a static analysis tool. In this approach, the static analysis generates verification conditions for an error, and the decision procedure attempts to determine if the error path is feasible or not. An example of this work is Check'n'Crash, which combines the ESC/Java prover with the POOC integer constraint solver [12].

The Synergy algorithm uses dynamic testing with symbolic execution to refine a program abstraction for property checking [18]. This line of work follows on earlier model-checking efforts such as SLAM and BLAST [2] [4]. Given an abstraction, Synergy generates a test case designed to refine the abstraction and uses the path condition of the concrete path executed by the test case. We differ in that we begin with an instrumented example program run and then attempt to generate test cases; our approach does not provide the same guarantees as Synergy, but it does allow us to leverage user-generated or fuzz test inputs.

Decision procedures have also been employed for understanding the behavior of malware. Seshia et al. use the UCLID decision procedure to detect code with semantics equivalent to a malware signature; in particular, their work detects and eliminates code sequences that act as no-ops [10]. Sweeper uses STP to find "trigger-based behavior" in malware [37]. The Replayer tool also uses STP, not to understand malware, but to aid in automatically analyzing binaries to discover characteristics of a protocol necessary for replaying a conversation [30].

Dynamic taint tracking has been previously implemented in Valgrind in the TaintCheck tool for the purposes of detecting attack by malware [31]. LIFT is a highly optimized implementation of taint tracking using DynamoRIO [33]. The Minos project proposed hardware support for enforcing an information flow security policy and gave a prototype implementation using the Bochs full-system emulator [11]. Recent versions of Perl also provide a taint mode for marking certain inputs as untrusted. These projects focus on detecting and preventing malicious behavior at runtime. Garfinkel et al. used dynamic taint tracking to understand the lifetime of sensitive data, such as passwords, in commodity operating systems [9]. Our approach, in contrast, uses taint analysis to prune the

constraints submitted to our solver instead of trying to detect run-time malicious behavior.

Our approach to type inference by solving type constraints is similar to that proposed by Mycroft [29]. His approach, however, was aimed at providing information for decompilation and reverse engineering of a program, and so used a significantly different type system. Xu, Miller, and Reps proposed a method for statically checking type safety of machine code, given typestate annotations on the program inputs [40]. In contrast, we work without such annotations, and we do not aim at guarantees that the program is type-safe. Instead, we attempt to generate inputs that exhibit a possible typing failure. Loginov et al. describe run-time type checking for debugging; the high-level idea of using type information to detect possible bad program behavior is similar, but they focus on providing warnings to a programmer while we focus on using type information to drive a decision procedure [23].

Besides STP, there are many other decision procedures, such as Yices and UCLID [14, 20]. Many of these take part in the annual SMT-LIB competition, which includes test cases from both hardware and software verification applications [36]. Automatic theorem provers have also been widely applied to program analysis, such as the Microsoft Zap prover [1].

We mentioned above that one approach to improving guest execution speed would be to instrument a program to retain side effects only, and defer the translation to STP formulas. The Amber debugger uses Valgrind to perform such side effect recording; O'Callahan reports that Amber can record a Mozilla session in roughly half an hour [32]. The Nirvana system from Microsoft Research integrates such tracing with a binary instrumentation framework roughly comparable to Valgrind [5]. Besides Valgrind and Nirvana, there are also other dynamic binary instrumentation frameworks, such as DynamoRIO, Vulcan, and Pin [7] [15] [24]. We chose Valgrind because the VEX intermediate representation is fairly close to STP's input language; in addition, the entire source of Valgrind is available, which we have found helpful in troubleshooting interactions between Catchconv and the Valgrind core.

We are agnostic about the source of the example input required by our tool, but we anticipate using our work on the output of fuzz testing tools. Our hypothesis is that using fuzz testers to generate the concrete input will allow us to reach deeper bugs than direct exploration with symbolic execution alone. Fuzz testing has received a great deal of attention recently, having been successfully employed in the Month of Browser Bugs and Month of Kernel Bugs [28, 22]. The term "fuzzing" is due to Miller et al., who ran early tests of UNIX and Windows utilities, and more recently tests of the MacOS GUI [27] [25] [26]. DeMott surveys recent work on fuzz testing, including the autodafe fuzzer, which uses `libgdb` to instrument functions of interest and adjust fuzz testing based on those functions' arguments [13, 38].

8 Conclusion

Our tool successfully generates path conditions and `QUERY` statements for synthetic test cases and some small but real programs. Synthesizing test inputs, however, for all but the smallest cases is a problem because of the large size of the resulting formulas. While the STP decision procedure we use has made and continues to make major improvements on our test cases, scaling up to large programs appears out of reach with the current implementation of formula generation. Fortunately, there are several directions available for pruning formulas before passing them to the decision procedure. Therefore, at this point it is too early to determine whether our approach will be a success. Future work will tell if our approach can combine fuzz testing and symbolic execution to perform better testing than either alone.

9 Acknowledgments

We thank Sanjit Seshia, Doug Tygar, and David Aldous for feedback and advice during this project, and for serving on the first author's qualifying exam committee. We are indebted to Vijay Ganesh and David Dill for the use of the STP tool, and for numerous improvements to STP that dramatically increased performance on our test cases. We thank Jason Waddle, DeanM, and Daniel Wilkerson for helpful discussions. We thank posters to the `fuzzing@whitestar.linuxbox.org` mailing list, including Jared DeMott, Disco Jonny, and Ari Takanen, for discussions on fuzzing tradeoffs. David Molnar was supported by an NSF Graduate Research Fellowship.

References

1. T. Ball, S.K. Lahiri, and M. Musuvathi. Zap: Automated theorem proving for software analysis. In *12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, 2005.
2. T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN2001 Workshop on Model Checking of Software, LNCS 2057*, 2001.
3. Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the* 16^{th} *International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, July 2004. Boston, Massachusetts.
4. D. Beyer, A. Chlipala, T. Henzinger, R. Jhala, and R. Majumdar. The BLAST query language for software verification. In *Static Analysis Symposium SAS*, 2004.
5. S. Bhansali, W. Chen, S. de Jong, A. Edwards, R. Murray, M. Dirnic, D. Mihocka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE*, 2006. `http://research.microsoft.com/manuvir/papers/instruction_level_tracing_VEE06.pdf`.
6. blexim. Basic integer overflows. *Phrack*, 0x0b(0x3c), 2002. `http://www.phrack.org/archives/60/p60-0x0a.txt`.

7. D. Bruening, V. Kiriansky, T. Garnett, E. Duesterwald, and S. Amarasinghe. The DynamoRIO collaboration, 2007. http://www.cag.lcs.mit.edu/dynamorio/.
8. Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, New York, NY, USA, 2006. ACM Press.
9. Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proc. 13th USENIX Security Symposium*, August 2004.
10. Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*, 2005.
11. J.R. Crandall and F.T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *IEEE Symposium on Microarchitecture*, December 2004.
12. C. Csallner and Y. Smaragdakis. Check'n'crash: Combining static checking and testing. In *International Conference on Software Engineering ICSE*, 2005.
13. J. DeMott. The evolving art of fuzzing. In *DEF CON 14*, 2006. http://www.appliedsec.com/files/The_Evolving_Art_of_Fuzzing.odp.
14. B. Dutertre and L. de Moura. A fast linear-arithmetic solver for dpll(t)*. In *Computer Aided Verification (CAV)*, 2006.
15. A. Edwards, A. Srivastava, and H. Vo. Vulcan: Binary transformation in a distributed environment, 2001. MSR-TR-2001-50 http://research.microsoft.com/research/pubs/view.aspx?msr_tr_id=MSR-TR-2001-50.
16. V. Ganesh and D. Dill. STP: A decision procedure for bitvectors and arrays. Under submission, 2007. http://theory.stanford.edu/~vganesh/stp.html.
17. P. Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *PLDI*, 2005.
18. B. Gulavani, T. Henzinger, Y. Kanan, A. Nori, and S. Rajamani. Synergy: A new algorithm for property checking. In *Proceedings of the 14th Annual Symposium on Foundations of Software Engineering (FSE)*, 2006.
19. James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
20. S.K. Lahiri and S. A. Seshia. The UCLID decision procedure. In *CAV*, 2004.
21. E. Larson and T. Austin. High-coverage detection of input-related security faults. In *Usenix Security*, 2003.
22. LMH. Month of kernel bugs, November 2006. http://projects.info-pull.com/mokb/.
23. A. Loginov, S.H. Yong, S.Horwitz, and T. Reps. Debugging via run-time type checking. In *Proc. of FASE 2001: Fundamental Approaches to Softw. Eng*, 2001. http://www.cs.wisc.edu/wpis/papers/fase01.ps.
24. Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.
25. Barton Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin, 1995.
26. Barton P. Miller, Gregory Cooksey, and Fredrick Moore. An empirical study of the robustness of macos applications using random testing. In *RT '06: Proceedings*

of the 1st international workshop on Random testing, pages 46–54, New York, NY, USA, 2006. ACM Press.
27. Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the Association for Computing Machinery*, 33(12):32–44, 1990.
28. H.D. Moore. Month of browser bugs, July 2006. `http://browserfun.blogspot.com/`.
29. A. Mycroft. Type-based decompilation. In *ESOP Springer-Verlag Lecture Notes in Computer Science vol. 1576*, 1999.
30. James Newsome, David Brumley, Jason Franklin, and Dawn Song. Replayer: Automatic protocol replay by binary analysis. In *In the Proceedings of the* 13^{th} *ACM Conference on Computer and and Communications Security (CCS)*, 2006.
31. James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.
32. Rob O'Callahan. A new approach to debugging, December 2006. `http://weblogs.mozillazine.org/roc/archives/2006/12/introducing_amb.html`.
33. F. Qin, H. Chen, Z. Li, Y. Zhou, H. Kim, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting general security attacks. In *MICRO*, 2006.
34. K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE*, 2005.
35. J. Seward and N. Nethercote. Using valgrind to detect undefined memory errors with bit precision. In *Proceedings of the USENIX Annual Technical Conference*, 2005. `http://www.valgrind.org/docs/memcheck2005.pdf`.
36. SMT-LIB. SMT-LIB web site, 2007. `http://combination.cs.uiowa.edu/smtlib/`.
37. J. Tucek, J. Newsome, S. Lu, D. Burmley, Y. Zhou, and D. Song. Sweeper: A lightweight end-to-end system for defending against fast worms. In *EuroSys : European Workshop on Operating Systems*, 2007.
38. M. Vuganoux. Autodafe - an act of software torture, 2005. Chaos Communications Congress, `http://autodafe.sourceforge.net/`.
39. David A. Wheeler. SLOCCount, 2004.
40. Zhichen Xu, Barton P. Miller, and Thomas Reps. Safety checking of machine code. *ACM SIGPLAN Notices*, 35(5):70–82, 2000.