Dynamic Test Generation To Find Integer Bugs in x86 Binary Linux Programs

David Molnar
UC Berkeley

Xue Cong Li
UC Berkeley

David A. Wagner
UC Berkeley

Abstract

Recently, integer bugs, including integer overflow, width conversion, and signed/unsigned conversion errors, have risen to become a common root cause for serious security vulnerabilities. We introduce new methods for discovering integer bugs using dynamic test generation on x86 binaries, and we describe key design choices in efficient symbolic execution of such programs. We implemented our methods in a prototype tool *SmartFuzz*, which we use to analyze Linux x86 binary executables. We also created a reporting service, `metafuzz.com`, to aid in triaging and reporting bugs found by SmartFuzz and the black-box fuzz testing tool `zzuf`. We report on experiments applying these tools to a range of software applications, including the `mplayer` media player, the `exiv2` image metadata library, and ImageMagick `convert`. We also report on our experience using SmartFuzz, `zzuf`, and `metafuzz.com` to perform testing at scale with the Amazon Elastic Compute Cloud (EC2). To date, the `metafuzz.com` site has recorded more than $2,614$ test runs, comprising $2,361,595$ test cases. Our experiments found approximately 77 total distinct bugs in 864 compute hours, costing us an average of $\$2.24$ per bug at current EC2 rates. We quantify the overlap in bugs found by the two tools, and we show that SmartFuzz finds bugs missed by `zzuf`, including one program where SmartFuzz finds bugs but `zzuf` does not.

1 Introduction

Integer overflow bugs recently became the second most common bug type in security advisories from OS vendors [10]. Unfortunately, traditional static and dynamic analysis techniques are poorly suited to detecting integer-related bugs. In this paper, we argue that *dynamic test generation* is better suited to finding such bugs, and we develop new methods for finding a broad class of integer bugs with this approach. We have implemented these methods in a new tool, *SmartFuzz*, that analyzes traces from commodity Linux x86 programs.

Integer bugs result from a mismatch between machine arithmetic and mathematical arithmetic. For example, machine arithmetic has bounded precision; if an expression has a value greater than the maximum integer that can be represented, the value wraps around to fit in machine precision. This can cause the value stored to be smaller than expected by the programmer. If, for example, a wrapped value is used as an argument to `malloc`, the result is an object that is smaller than expected, which can lead to a buffer overflow later if the programmer is not careful. This kind of bug is often known as an integer overflow bug. In Section 2 we describe two other classes of integer bugs: *width conversions*, in which converting from one type of machine integer to another causes unexpected changes in value, and *signed/unsigned conversions*, in which a value is treated as both a signed and an unsigned integer. These kinds of bugs are pervasive and can, in many cases, cause serious security vulnerabilities. Therefore, eliminating such bugs is important for improving software security.

While new code can partially or totally avoid integer bugs if it is constructed appropriately [19], it is also important to find and fix bugs in legacy code. Previous approaches to finding integer bugs in legacy code have focused on static analysis or runtime checks. Unfortunately, existing static analysis algorithms for finding integer bugs tend to generate many false positives, because it is difficult to statically reason about integer values with sufficient precision. Alternatively, one can insert runtime checks into the application to check for overflow or non-value-preserving width conversions, and raise an exception if they occur. One problem with this approach is that many overflows are benign and harmless. Throwing an exception in such cases prevents the application from functioning and thus causes false positives. Furthermore, occasionally the code intentionally relies upon overflow semantics; e.g., cryptographic code or fast hash functions. Such code is often falsely flagged by static analysis or runtime checks. In summary, both static analysis and

runtime checking tend to suffer from either many false positives or many missed bugs.

In contrast, *dynamic test generation* is a promising approach for avoiding these shortcomings. Dynamic test generation, a technique introduced by Godefroid et al. and Engler et al. [13, 7], uses *symbolic execution* to generate new test cases that expose specifically targeted behaviors of the program. Symbolic execution works by collecting a set of constraints, called the *path condition*, that model the values computed by the program along a single path through the code. To determine whether there is any input that could cause the program to follow that path of execution and also violate a particular assertion, we can add to the path condition a constraint representing that the assertion is violated and feed the resulting set of constraints to a solver. If the solver finds any solution to the resulting constraints, we can synthesize a new test case that will trigger an assertion violation. In this way, symbolic execution can be used to discover test cases that cause the program to behave in a specific way.

Our main approach is to use symbolic execution to construct test cases that trigger arithmetic overflows, non-value-preserving width conversions, or dangerous signed/unsigned conversions. Then, we run the program on these test cases and use standard tools that check for buggy behavior to recognize bugs. We only report test cases that are verified to trigger incorrect behavior by the program. As a result, we have confidence that all test cases we report are real bugs and not false positives.

Others have previously reported on using dynamic test generation to find some kinds of security bugs [8, 15]. The contribution of this paper is to show how to extend those techniques to find integer-related bugs. We show that this approach is effective at finding many bugs, without the false positives endemic to prior work on static analysis and runtime checking.

The ability to eliminate false positives is important, because false positives are time-consuming to deal with. In slogan form: false positives in static analysis waste the programmer's time; false positives in runtime checking waste the end user's time; while false positives in dynamic test generation waste the tool's time. Because an hour of CPU time is much cheaper than an hour of a human's time, dynamic test generation is an attractive way to find and fix integer bugs.

We have implemented our approach to finding integer bugs in *SmartFuzz*, a tool for performing symbolic execution and dynamic test generation on Linux x86 applications. SmartFuzz works with binary executables directly, and does not require or use access to source code. Working with binaries has several advantages, most notably that we can generate tests directly from shipping binaries. In particular, we do not need to modify the build process for a program under test, which has been a pain point for static analysis tools [9]. Also, this allows us to perform whole-program analysis: we can find bugs that arise due to interactions between the application and libraries it uses, even if we don't have source code for those libraries. Of course, working with binary traces introduces special challenges, most notably the sheer size of the traces and the lack of type information that would be present in the source code. We discuss the challenges and design choices in Section 4.

In Section 5 we describe the techniques we use to generate test cases for integer bugs in dynamic test generation. We discovered that these techniques find many bugs, too many to track manually. To help us prioritize and manage these bug reports and streamline the process of reporting them to developers, we built *Metafuzz*, a web service for tracking test cases and bugs (Section 6). Metafuzz helps minimize the amount of human time required to find high-quality bugs and report them to developers, which is important because human time is the most expensive resource in a testing framework. Finally, Section 7 presents an empirical evaluation of our techniques and discusses our experience with these tools.

The contributions of this paper are the following:

- We design novel algorithms for finding signed/unsigned conversion vulnerabilities using symbolic execution. In particular, we develop a novel type inference approach that allows us to detect which values in an x86 binary trace are used as signed integers, unsigned integers, or both. We discuss challenges in scaling such an analysis to commodity Linux media playing software and our approach to these challenges.

- We extend the range of integer bugs that can be found with symbolic execution, including integer overflows, integer underflows, width conversions, and signed/unsigned conversions. No prior symbolic execution tool has included the ability to detect all of these kinds of integer vulnerabilities.

- We implement these methods in *SmartFuzz*, a tool for symbolic execution and dynamic test generation of x86 binaries on Linux. We describe key challenges in symbolic execution of commodity Linux software, and we explain design choices in SmartFuzz motivated by these challenges.

- We report on the bug finding performance of SmartFuzz and compare SmartFuzz to the `zzuf` black box fuzz testing tool. The `zzuf` tool is a simple, yet effective, *fuzz testing* program which randomly mutates a given seed file to find new test inputs, without any knowledge or feedback from the target program. We have tested a broad range of commodity Linux software, including the media players

mplayer and ffmpeg, the ImageMagick convert tool, and the exiv2 TIFF metadata parsing library. This software comprises over one million lines of source code, and our test cases result in symbolic execution of traces that are millions of x86 instructions in length.

- We identify challenges with reporting bugs at scale, and introduce several techniques for addressing these challenges. For example, we present evidence that a simple stack hash is not sufficient for grouping test cases to avoid duplicate bug reports, and then we develop a *fuzzy stack hash* to solve these problems. Our experiments find approximately 77 total distinct bugs in 864 compute hours, giving us an average cost of $2.24 per bug at current Amazon EC2 rates. We quantify the overlap in bugs found by the two tools, and we show that SmartFuzz finds bugs missed by zzuf, including one program where SmartFuzz finds bugs but zzuf does not.

Between June 2008 and November 2008, Metafuzz has processed over 2,614 test runs from both SmartFuzz and the zzuf black box fuzz testing tool [16], comprising 2,361,595 test cases. To our knowledge, this is the largest number of test runs and test cases yet reported for dynamic test generation techniques. We have released our code under the GPL version 2 and BSD licenses[1]. Our vision is a service that makes it easy and inexpensive for software projects to find integer bugs and other serious security relevant code defects using dynamic test generation techniques. Our work shows that such a service is possible for a large class of commodity Linux programs.

2 Integer Bugs

We now describe the three main classes of integer bugs we want to find: integer overflow/underflow, width conversions, and signed/unsigned conversion errors [2]. All three classes of bugs occur due to the mismatch between machine arithmetic and arithmetic over unbounded integers.

Overflow/Underflow. Integer overflow (and underflow) bugs occur when an arithmetic expression results in a value that is larger (or smaller) than can be represented by the machine type. The usual behavior in this case is to silently "wrap around," e.g. for a 32-bit type, reduce the value modulo 2^{32}. Consider the function badalloc in Figure 1. If the multiplication sz * n overflows, the allocated buffer may be smaller than expected, which can lead to a buffer overflow later.

Width Conversions. Converting a value of one integral type to a wider (or narrower) integral type which has a different range of values can introduce *width conversion* bugs. For instance, consider badcpy in Figure 1. If the first parameter is negative, the conversion from Int16 to UInt32 will trigger sign-extension, causing m to be very large and likely leading to a buffer overflow. Because memcpy's third argument is declared to have type size_t (which is an unsigned integer type), even if we passed n directly to memcpy the implicit conversion would still make this buggy. Width conversion bugs can also arise when converting a wider type to a narrower type.

```
char *badalloc(int sz, int n) {
  return (char *) malloc(sz * n);
}
void badcpy(Int16 n, char *p, char *q) {
  UInt32 m = n;
  memcpy(p, q, m);
}
void badcpy2(int n, char *p, char *q) {
  if (n > 800)
     return;
  memcpy(p, q, n);
}
```

Figure 1: Examples of three types of integer bugs.

Signed/Unsigned Conversion. Lastly, converting a signed integer type to an unsigned integer type of the same width (or vice versa) can introduce bugs, because this conversion can change a negative number to a large positive number (or vice versa). For example, consider badcpy2 in Figure 1. If the first parameter n is a negative integer, it will pass the bounds check, then be promoted to a large unsigned integer when passed to memcpy. memcpy will copy a large number of bytes, likely leading to a buffer overflow.

3 Related Work

An earlier version of SmartFuzz and the Metafuzz web site infrastructure described in this paper were used for previous work that compares dynamic test generation with black-box fuzz testing by different authors [1]. That previous work does not describe the SmartFuzz tool, its design choices, or the Metafuzz infrastructure in detail. Furthermore, this paper works from new data on the effectiveness of SmartFuzz, except for an anecdote in our "preliminary experiences" section. We are not aware of other work that directly compares dynamic test generation with black-box fuzz testing on a scale similar to ours.

The most closely related work on integer bugs is Godefroid et al. [15], who describe dynamic test generation with bug-seeking queries for integer overflow, underflow, and some narrowing conversion errors in the context of the SAGE tool. Our work looks at a wider

[1] http://www.sf.net/projects/catchconv

range of narrowing conversion errors, and we consider signed/unsigned conversion while their work does not. The EXE and KLEE tools also use integer overflow to prioritize different test cases in dynamic test generation, but they do not break out results on the number of bugs found due to this heuristic [8, 6]. The KLEE system also focuses on scaling dynamic test generation, but in a different way. While we focus on a few "large" programs in our results, KLEE focuses on high code coverage for over 450 smaller programs, as measured by trace size and source lines of code. These previous works also do not address the problem of type inference for integer types in binary traces.

IntScope is a static binary analysis tool for finding integer overflow bugs [28]. IntScope translates binaries to an intermediate representation, then it checks lazily for potentially harmful integer overflows by using symbolic execution for data that flows into "taint sinks" defined by the tool, such as memory allocation functions. SmartFuzz, in contrast, *eagerly* attempts to generate new test cases that cause an integer bug at the point in the program where such behavior could occur. This difference is due in part to the fact that IntScope reports errors to a programmer directly, while SmartFuzz filters test cases using a tool such as `memcheck`. As we argued in the Introduction, such a filter allows us to employ aggressive heuristics that may generate many test cases. Furthermore, while IntScope renders signed and unsigned comparisons in their intermediate representation by using hints from the x86 instruction set, they do not explicitly discuss how to use this information to perform type inference for signed and unsigned types, nor do they address the issue of scaling such inference to traces with millions of instructions. Finally, IntScope focuses only on integer overflow errors, while SmartFuzz covers underflow, narrowing conversion, and signed/unsigned conversion bugs in addition.

The dynamic test generation approach we use was introduced by Godefroid et al. [13] and independently by Cadar and Engler [7]. The SAGE system by Godefroid et al. works, as we do, on x86 binary programs and uses a generational search, but SAGE makes several different design choices we explain in Section 4. Lanzi et al. propose a design for dynamic test generation of x86 binaries that uses static analysis of loops to assist the solver, but their implementation is preliminary [17]. KLEE, in contrast, works with the intermediate representation generated by the Low-Level Virtual Machine target for `gcc` [6]. Larson and Austin applied symbolic range analysis to traces of programs to look for potential buffer overflow attacks, although they did not attempt to synthesize crashing inputs [18]. The BitBlaze [5] infrastructure of Song et al. also performs symbolic execution of x86 binaries, but their focus is on malware and signature generation, not on test generation.

Other approaches to integer bugs include static analysis and runtime detection. The Microsoft Prefast tool uses static analysis to warn about intraprocedural integer overflows [21]. Both Microsoft Visual C++ and `gcc` can add runtime checks to catch integer overflows in arguments to `malloc` and terminate a program. Brumley et al. provide rules for such runtime checks and show they can be implemented with low overhead on the x86 architecture by using jumps conditioned on the overflow bit in EFLAGS [4]. Both of these approaches fail to catch signed/unsigned conversion errors. Furthermore, both static analysis and runtime checking for overflow will flag code that is correct but relies on overflow semantics, while our approach only reports test cases in case of a crash or a Valgrind error report.

Blexim gives an introduction to integer bugs [3]. Fuzz testing has received a great deal of attention since its original introduction by Miller et al [22]. Notable public demonstrations of fuzzing's ability to find bugs include the Month of Browser Bugs and Month of Kernel Bugs [23, 20]. 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 [11, 27].

Our Metafuzz infrastructure also addresses issues not treated in previous work on test generation. First, we make *bug bucketing* a first-class problem and we introduce a *fuzzy stack hash* in response to developer feedback on bugs reported by Metafuzz. The SAGE paper reports bugs by stack hash, and KLEE reports on using the line of code as a bug bucketing heuristic, but we are not aware of other work that uses a fuzzy stack hash. Second, we report techniques for reducing the amount of human time required to process test cases generated by fuzzing and improve the quality of our error reports to developers; we are not aware of previous work on this topic. Such techniques are vitally important because human time is the most expensive part of a test infrastructure. Finally, Metafuzz uses on-demand computing with the Amazon Elastic Compute Cloud, and we explicitly quantify the cost of each bug found, which was not done in previous work.

4 Dynamic Test Generation

We describe the architecture of *SmartFuzz*, a tool for dynamic test generation of x86 binary programs on Linux. Dynamic test generation on x86 binaries—without access to source code—raises special challenges. We discuss these challenges and motivate our fundamental design choices.

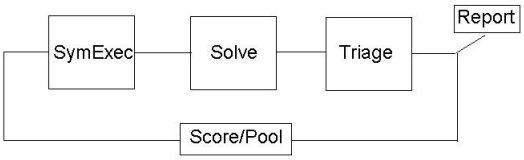

Figure 2: Dynamic test generation includes four stages: symbolic execution, solving to obtain new test cases, then triage to determine whether to report a bug or score the test case for addition to the pool of unexplored test cases.

4.1 Architecture

The SmartFuzz architecture is as follows: First, we add one or more test cases to a pool. Each test case in the pool receives a score given by the number of new basic blocks seen when running the target program on the test case. By "new" we mean that the basic block has not been observed while scoring any previous test case; we identify basic blocks by the instruction pointer of their entry point.

In each iteration of test generation, we choose a high-scoring test case, execute the program on that input, and use symbolic execution to generate a set of constraints that record how each intermediate value computed by the program relates to the inputs in the test case. SmartFuzz implements the symbolic execution and scoring components using the Valgrind binary analysis framework, and we use STP [12] to solve constraints.

For each symbolic branch, SmartFuzz adds a constraint that tries to force the program down a different path. We then query the constraint solver to see whether there exists any solution to the resulting set of constraints; if there is, the solution describes a new test case. We refer to these as *coverage queries* to the constraint solver.

SmartFuzz also injects constraints that are satisfied if a condition causing an error or potential error is satisfied (e.g., to force an arithmetic calculation to overflow). We then query the constraint solver; a solution describes a test case likely to cause an error. We refer to these as *bug-seeking queries* to the constraint solver. Bug-seeking queries come in different *types*, depending on the specific error they seek to exhibit in the program.

Both coverage and bug-seeking queries are explored in a *generational search* similar to the SAGE tool [14]. Each query from a symbolic trace is solved in turn, and new test cases created from successfully solved queries. A single symbolic execution therefore leads to many coverage and bug-seeking queries to the constraint solver, which may result in many new test cases.

We *triage* each new test case as it is generated, i.e. we determine if it exhibits a bug. If so, we report the bug; otherwise, we add the test case to the pool for scoring and possible symbolic execution. For triage, we use Valgrind `memcheck` on the target program with each test case, which is a tool that observes concrete execution looking for common programming errors [26]. We record any test case that causes the program to crash or triggers a `memcheck` warning.

We chose `memcheck` because it checks a variety of properties, including reads and writes to invalid memory locations, memory leaks, and use of uninitialized values. Re-implementing these analyses as part of the SmartFuzz symbolic execution tool would be wasteful and error-prone, as the `memcheck` tool has had the benefit of multiple years of use in large-scale projects such as Firefox and OpenOffice. The `memcheck` tool is known as a tool with a low false positive rate, as well, making it more likely that developers will pay attention to bugs reported by `memcheck`. Given a `memcheck` error report, developers do not even need to know that associated test case was created by SmartFuzz.

We do not attempt to classify the bugs we find as exploitable or not exploitable, because doing so by hand for the volume of test cases we generate is impractical. Many of the bugs found by `memcheck` are memory safety errors, which often lead to security vulnerabilities. Writes to invalid memory locations, in particular, are a red flag. Finally, to report bugs we use the Metafuzz framework described in Section 6.

4.2 Design Choices

Intermediate Representation. The sheer size and complexity of the x86 instruction set poses a challenge for analyzing x86 binaries. We decided to translate the underlying x86 code on-the-fly to an intermediate representation, then map the intermediate representation to symbolic formulas. Specifically, we used the Valgrind binary instrumentation tool to translate x86 instructions into VEX, the Valgrind intermediate representation [24]. The BitBlaze system works similarly, but with a different intermediate representation [5]. Details are available in an extended version of this paper[2].

Using an intermediate representation offers several advantages. First, it allows for a degree of platform independence: though we support only x86 in our current tool, the VEX library also supports the AMD64 and PowerPC instruction sets, with ARM support under active development. Adding support for these additional architectures requires only adding support for a small number of additional VEX instructions, not an entirely new instruction set from scratch. Second, the VEX library generates IR that satisfies the single static assignment property and

[2] `http://www.cs.berkeley.edu/~dmolnar/usenix09-full.pdf`

performs other optimizations, which makes the translation from IR to formulas more straightforward. Third, and most importantly, this choice allowed us to outsource the pain of dealing with the minutae of the x86 instruction set to the VEX library, which has had years of production use as part of the Valgrind memory checking tool. For instance, we don't need to explicitly model the EFLAGS register, as the VEX library translates it to boolean operations. The main shortcoming with the VEX IR is that a single x86 instruction may expand to five or more IR instructions, which results in long traces and correspondingly longer symbolic formulas.

Online Constraint Generation. SmartFuzz uses *online* constraint generation, in which constraints are generated while the program is running. In contrast, SAGE (another tool for dynamic test generation) uses *offline* constraint generation, where the program is first traced and then the trace is replayed to generate constraints [14]. Offline constraint generation has several advantages: it is not sensitive to concurrency or nondeterminism in system calls; tracing has lower runtime overhead than constraint generation, so can be applied to running systems in a realistic environment; and, this separation of concerns makes the system easier to develop and debug, not least because trace replay and constraint generation is reproducible and deterministic. In short, offline constraint generation has important software engineering advantages.

SmartFuzz uses online constraint generation primarily because, when the SmartFuzz project began, we were not aware of an available offline trace-and-replay framework with an intermediate representation comparable to VEX. Today, O'Callahan's `chronicle-recorder` could provide a starting point for a VEX-based offline constraint generation tool [25].

Memory Model. Other symbolic execution tools such as EXE and KLEE model memory as a set of symbolic arrays, with one array for each allocated memory object. We do not. Instead, for each load or store instruction, we first concretize the memory address before accessing the symbolic heap. In particular, we keep a map M from concrete memory addresses to symbolic values. If the program reads from concrete address `a`, we retrieve a symbolic value from $M(\texttt{a})$. Even if we have recorded a symbolic expression a associated with this address, the symbolic address is ignored. Note that the value of `a` is known at constraint generation time and hence becomes (as far as the solver is concerned) a constant. Store instructions are handled similarly.

While this approach sacrifices precision, it scales better to large traces. We note that the SAGE tool adopts a similar memory model. In particular, concretizing addresses generates symbolic formulas that the constraint solver can solve much more efficiently, because the solver does not need to reason about aliasing of pointers.

Only Tainted Data is Symbolic. We track the taint status of every byte in memory. As an optimization, we do not store symbolic information for untainted memory locations, because by definition untainted data is not dependent upon the untrusted inputs that we are trying to vary. We have found that only a tiny fraction of the data processed along a single execution path is tainted. Consequently, this optimization greatly reduces the size of our constraint systems and reduces the memory overhead of symbolic execution.

Focus on Fuzzing Files. We decided to focus on single-threaded programs, such as media players, that read a file containing untrusted data. Thus, a test case is simply the contents of this file, and SmartFuzz can focus on generating candidate files. This simplifies the symbolic execution and test case generation infrastructure, because there are a limited number of system calls that read from this file, and we do not need to account for concurrent interactions between threads in the same program. We know of no fundamental barriers, however, to extending our approach to multi-threaded and network-facing programs.

Our implementation associates a symbolic input variable with each byte of the input file. As a result, SmartFuzz cannot generate test cases with more bytes than are present in the initial seed file.

Multiple Cooperating Analyses. Our tool is implemented as a series of independent cooperating analyses in the Valgrind instrumentation framework. Each analysis adds its own instrumentation to a basic block during translation and exports an interface to the other analyses. For example, the instrumentation for tracking taint flow, which determines the IR instructions to treat as symbolic, exports an interface that allows querying whether a specific memory location or temporary variable is symbolic. A second analysis then uses this interface to determine whether or not to output STP constraints for a given IR instruction.

The main advantage of this approach is that it makes it easy to add new features by adding a new analysis, then modifying our core constraint generation instrumentation. Also, this decomposition enabled us to extract our taint-tracking code and use it in a different project with minimal modifications, and we were able to implement the binary type inference analysis described in Section 5, replacing a different earlier version, without changing our other analyses.

Optimize in Postprocessing. Another design choice was to output constraints that are as "close" as possible to the intermediate representation, performing only limited optimizations on the fly. For example, we implement the "related constraint elimination," as introduced by tools

such as EXE and SAGE [8, 14], as a post-processing step on constraints created by our tool. We then leave it up to the solver to perform common subexpression elimination, constant propagation, and other optimizations. The main benefit of this choice is that it simplifies our constraint generation. One drawback of this choice is that current solvers, including STP, are not yet capable of "remembering" optimizations from one query to the next, leading to redundant work on the part of the solver. The main drawback of this choice, however, is that while after optimization each individual query is small, the total symbolic trace containing all queries for a program can be several gigabytes. When running our tool on a 32-bit host machine, this can cause problems with maximum file size for a single file or maximum memory size in a single process.

5 Techniques for Finding Integer Bugs

We now describe the techniques we use for finding integer bugs.

Overflow/Underflow. For each arithmetic expression that could potentially overflow or underflow, we emit a constraint that is satisfied if the overflow or underflow occurs. If our solver can satisfy these constraints, the resulting input values will likely cause an underflow or overflow, potentially leading to unexpected behavior.

Width Conversions. For each conversion between integer types, we check whether it is possible for the source value to be outside the range of the target value by adding a constraint that's satisfied when this is the case and then applying the constraint solver. For conversions that may sign-extend, we use the constraint solver to search for a test case where the high bit of the source value is nonzero.

Signed/Unsigned Conversions. Our basic approach is to try to reconstruct, from the x86 instructions executed, signed/unsigned type information about all integral values. This information is present in the source code but not in the binary, so we describe an algorithm to infer this information automatically.

Consider four types for integer values: "Top," "Signed," "Unsigned," or "Bottom." Here, "Top" means the value has not been observed in the context of a signed or unsigned integer; "Signed" means that the value has been used as a signed integer; "Unsigned" means the value has been used as an unsigned integer; and "Bottom" means that the value has been used inconsistently as both a signed and unsigned integer. These types form a four-point lattice. Our goal is to find symbolic program values that have type "Bottom." These values are candidates for signed/unsigned conversion errors. We then attempt to synthesize an input that forces these values to be negative.

We associate every instance of every temporary variable in the Valgrind intermediate representation with a type. Every variable in the program starts with type Top. During execution we add *type constraints* to the type of each value. For x86 binaries, the sources of type constraints are signed and unsigned comparison operators: e.g., a signed comparison between two values causes both values to receive the "Signed" type constraint. We also add unsigned type constraints to values used as the length argument of `memcpy` function, which we can detect because we know the calling convention for x86 and we have debugging symbols for glibc. While the x86 instruction set has additional operations, such as `IMUL` that reveal type information about their operands, we do not consider these; this means only that we may incorrectly under-constrain the types of some values.

```
int main(int argc, char** argv) {
  char * p = malloc(800);
  char * q = malloc(800);
  int n;
  n = atol(argv[1]);
  if (n > 800)
      return;
  memcpy(p, q, n);
  return 0;
}
```

Figure 3: A simple test case for dynamic type inference and query generation. The signed comparison `n > 800` and unsigned `size_t` argument to `memcpy` assign the type "Bottom" to the value associated with `n`. When we solve for an input that makes `n` negative, we obtain a test case that reveals the error.

Any value that has received both a signed and unsigned type constraint receives the type Bottom. After adding a type constraint, we check to see if the type of a value has moved to Bottom. If so, we attempt to solve for an input which makes the value negative. We do this because negative values behave differently in signed and unsigned comparisons, and so they are likely to exhibit an error if one exists. All of this information is present in the trace without requiring access to the original program source code.

We discovered, however, that `gcc 4.1.2` inlines some calls to `memcpy` by transforming them to `rep movsb` instructions, even when the `-O` flag is not present. Furthermore, the Valgrind IR generated for the `rep movsb` instruction compares a decrementing counter variable to zero, instead of counting up and executing an unsigned comparison to the loop bound. As a result, on gcc 4.1.2 a call to `memcpy` does not cause its length argument to be marked as unsigned. To deal with this problem, we implemented a simple heuristic to detect the IR generated for `rep movsb` and emit the appropriate con-

straint. We verified that this heuristic works on a small test case similar to Figure 3, generating a test input that caused a segmentation fault.

A key problem is storing all of the information required to carry out type inference without exhausting available memory. Because a trace may have several million instructions, memory usage is key to scaling type inference to long traces. Furthermore, our algorithm requires us to keep track of the types of all values in the program, unlike constraint generation, which need concern itself only with tainted values. An earlier version of our analysis created a special "type variable" for each value, then maintained a map from IR locations to type variables. Each type variable then mapped to a type. We found that in addition to being hard to maintain, this analysis often led to a number of live type variables that scaled linearly with the number of executed IR instructions. The result was that our analysis ran out of memory when attempting to play media files in the `mplayer` media player.

To solve this problem, we developed a garbage-collected data structure for tracking type information. To reduce memory consumption, we use a union-find data structure to partition integer values into equivalence classes where all values in an equivalence class are required to have the same type. We maintain one type for each union-find equivalence class; in our implementation type information is associated with the representative node for that equivalence class. Assignments force the source and target values to have the same types, which is implemented by merging their equivalence classes. Updating the type for a value can be done by updating its representative node's type, with no need to explicitly update the types of all other variables in the equivalence class.

It turns out that this data structure is acyclic, due to the fact that VEX IR is in SSA form. Therefore, we use reference counting to garbage collect these nodes. In addition, we benefit from an additional property of the VEX IR: all values are either stored in memory, in registers, or in a temporary variable, and the lifetime of each temporary variable is implicitly limited to that of a single basic block. Therefore, we maintain a list of temporaries that are live in the current basic block; when we leave the basic block, the type information associated with all of those live temporaries can be deallocated. Consequently, the amount of memory needed for type inference at any point is proportional to the number of tainted (symbolic) variables that are live at that point—which is a significant improvement over the naive approach to type inference. The full version of this paper contains a more detailed specification of these algorithms[3].

[3] `http://www.cs.berkeley.edu/~dmolnar/usenix09-full.pdf`

6 Triage and Reporting at Scale

Both SmartFuzz and `zzuf` can produce hundreds to thousands of test cases for a single test run. We designed and built a web service, *Metafuzz*, to manage the volume of tests. We describe some problems we found while building Metafuzz and techniques to overcoming these problems. Finally, we describe the user experience with Metafuzz and bug reporting.

6.1 Problems and Techniques

The Metafuzz architecture is as follows: first, a Test Machine generates new test cases for a program and runs them locally. The Test Machine then determines which test cases exhibit bugs and sends these test cases to Metafuzz. The Metafuzz web site displays these test cases to the User, along with information about what kind of bug was found in which target program. The User can pick test cases of interest and download them for further investigation. We now describe some of the problems we faced when designing Metafuzz, and our techniques for handling them. Section 7 reports our experiences with using Metafuzz to manage test cases and report bugs.

Problem: Each test run generated many test cases, too many to examine by hand.

Technique: We used Valgrind's `memcheck` to automate the process of checking whether a particular test case causes the program to misbehave. `Memcheck` looks for memory leaks, use of uninitialized values, and memory safety errors such as writes to memory that was not allocated [26]. If `memcheck` reports an error, we save the test case. In addition, we looked for core dumps and non-zero program exit codes.

Problem: Even after filtering out the test cases that caused no errors, there were still many test cases that do cause errors.

Technique: The `metafuzz.com` front page is a HTML page listing all of the potential bug reports. showing all potential bug reports. Each test machine uploads information about test cases that trigger bugs to Metafuzz.

Problem: The machines used for testing had no long-term storage. Some of the test cases were too big to attach in e-mail or Bugzilla, making it difficult to share them with developers.

Technique: Test cases are uploaded directly to Metafuzz, providing each one with a stable URL. Each test case also includes the Valgrind output showing the Valgrind error, as well as the output of the program to `stdout` and `stderr`.

Problem: Some target projects change quickly. For example, we saw as many as four updates per day to the `mplayer` source code repository. Developers reject bug reports against "out of date" versions of the software.

Technique: We use the Amazon Elastic Compute Cloud (EC2) to automatically attempt to reproduce the bug

against the latest version of the target software. A button on the Metafuzz site spawns an Amazon EC2 instance that checks out the most recent version of the target software, builds it, and then attempts to reproduce the bug.

Problem: Software projects have specific reporting requirements that are tedious to implement by hand. For example, `mplayer` developers ask for a stack backtrace, disassembly, and register dump at the point of a crash.

Technique: Metafuzz automatically generates bug reports in the proper format from the failing test case. We added a button to the Metafuzz web site so that we can review the resulting bug report and then send it to the target software's bug tracker with a single click.

Problem: The same bug often manifests itself as many failing test cases. Reporting the same bug to developers many times wastes developer time.

Technique: We use the call stack to identify multiple instances of the same bug. Valgrind memcheck reports the call stack at each error site, as a sequence of instruction pointers. If debugging information is present, it also reports the associated filename and line number information in the source code.

Initially, we computed a *stack hash* as a hash of the sequence of instruction pointers in the backtrace. This has the benefit of not requiring debug information or symbols. Unfortunately, we found that a naive stack hash has several problems. First, it is sensitive to address space layout randomization (ASLR), because different runs of the same program may load the stack or dynamically linked libraries at different addresses, leading to different hash values for call stacks that are semantically the same. Second, even without ASLR, we found several cases where a single bug might be triggered at multiple call stacks that were similar but not identical. For example, a buggy function can be called in several different places in the code. Each call site then yields a different stack hash. Third, any slight change to the target software can change instruction pointers and thus cause the same bug to receive a different stack hash. While we do use the stack hash on the client to avoid uploading test cases for bugs that have been previously found, we found that we could not use stack hashes alone to determine if a bug report is novel or not.

To address these shortcomings, we developed a *fuzzy stack hash* that is forgiving of slight changes to the call stack. We use debug symbol information to identify the name of the function called, the line number in source code (excluding the last digit of the line number, to allow for slight changes in the code), and the name of the object file for each frame in the call stack. We then hash all of this information for the three functions at the top of the call stack.

The choice of the number of functions to hash determines the "fuzzyness" of the hash. At one extreme, we could hash all extant functions on the call stack. This would be similar to the classic stack hash and report many semantically same bugs in different buckets. On the other extreme, we could hash only the most recently called function. This fails in cases where two semantically different bugs both exhibit as a result of calling `memcpy` or some other utility function with bogus arguments. In this case, both call stacks would end with `memcpy` even though the bug is in the way the arguments are computed. We chose three functions as a trade-off between these extremes; we found this sufficient to stop further reports from the `mplayer` developers of duplicates in our initial experiences. Finding the best fuzzy stack hash is interesting future work; we note that the choice of bug bucketing technique may depend on the program under test.

While any fuzzy stack hash, including ours, may accidentally lump together two distinct bugs, we believe this is less serious than reporting duplicate bugs to developers. We added a post-processing step on the server that computes the fuzzy stack hash for test cases that have been uploaded to Metafuzz and uses it to coalesce duplicates into a single bug bucket.

Problem: Because Valgrind `memcheck` does not terminate the program after seeing an error, a single test case may give rise to dozens of Valgrind error reports. Two different test cases may share some Valgrind errors but not others.

Technique: First, we put a link on the Metafuzz site to a single test case for each bug bucket. Therefore, if two test cases share some Valgrind errors, we only use one test case for each of the errors in common. Second, when reporting bugs to developers, we highlight in the title the specific bugs on which to focus.

7 Results

7.1 Preliminary Experience

We used an earlier version of SmartFuzz and Metafuzz in a project carried out by a group of undergraduate students over the course of eight weeks in Summer 2008. When beginning the project, none of the students had any training in security or bug reporting. We provided a one-week course in software security. We introduced SmartFuzz, `zzuf`, and Metafuzz, then asked the students to generate test cases and report bugs to software developers. By the end of the eight weeks, the students generated over 1.2 million test cases, from which they reported over 90 bugs to software developers, principally to the `mplayer` project, of which 14 were fixed. For further details, we refer to their presentation [1].

7.2 Experiment Setup

Test Programs. Our target programs were `mplayer` version `SVN-r28403-4.1.2`, `ffmpeg` version

	mplayer	ffmpeg	exiv2	gzip	bzip2	convert
Coverage	2599	14535	1629	5906	12606	388
ConversionNot32	0	3787	0	0	0	0
Conversion32to8	1	26	740	2	10	116
Conversion32to16	0	16004	0	0	0	0
Conversion16Sto32	0	121	0	0	0	0
SignedOverflow	1544	37803	5941	24825	9109	49
SignedUnderflow	3	4003	48	1647	2840	0
UnsignedOverflow	1544	36945	4957	24825	9104	35
UnsignedUnderflow	0	0	0	0	0	0
MallocArg	0	24	0	0	0	0
SignedUnsigned	2568	21064	799	7883	17065	49

Figure 5: The number of each type of query for each test program after a single 24-hour run.

	Queries	Test Cases	Bugs
Coverage	588068	31121	19
ConversionNot32	4586	0	0
Conversion32to8	1915	1377	3
Conversion32to16	16073	67	4
Conversion16Sto32	206	0	0
SignedOverflow	167110	0	0
SignedUnderflow	20198	21	3
UnsignedOverflow	164155	9280	3
MallocArg	30	0	0
SignedUnsigned	125509	6949	5

Figure 6: The number of bugs found, by query type, over all test runs. The fourth column shows the number of distinct bugs found from test cases produced by the given type of query, as classified using our fuzzy stack hash.

`SVN-r16903`, `exiv2` version `SVN-r1735`, `gzip` version 1.3.12, `bzip2` version 1.0.5, and ImageMagick `convert` version $6.4.8 - 10$, which are all widely used media and compression programs. Table 4 shows information on the size of each test program. Our test programs are large, both in terms of source lines of code and trace lengths. The percentage of the trace that is symbolic, however, is small.

Test Platform. Our experiments were run on the Amazon Elastic Compute Cloud (EC2), employing a "small" and a "large" instance image with SmartFuzz, `zzuf`, and all our test programs pre-installed. At this writing, an EC2 small instance has 1.7 GB of RAM and a single-core virtual CPU with performance roughly equivalent to a 1GHz 2007 Intel Xeon. An EC2 large instance has 7 GB of RAM and a dual-core virtual CPU, with each core having performance roughly equivalent to a 1 GHz Xeon.

We ran all `mplayer` runs and `ffmpeg` runs on EC2 large instances, and we ran all other test runs with EC2 small instances. We spot-checked each run to ensure that instances successfully held all working data in memory during symbolic execution and triage without swapping to disk, which would incur a significant performance penalty. For each target program we ran SmartFuzz and `zzuf` with three seed files, for 24 hours per program per seed file. Our experiments took 288 large machine-hours and 576 small machine-hours, which at current EC2 prices of $0.10 per hour for small instances and $0.40 per hour for large instances cost $172.80.

Query Types. SmartFuzz queries our solver with the following types of queries: `Coverage`, `ConversionNot32`, `Conversion32to8`, `Conversion32to16`, `SignedOverflow`, `UnsignedOverflow`, `SignedUnderflow`, `UnsignedUnderflow`, `MallocArg`, and `SignedUnsigned`. `Coverage` queries refer to queries created as part of the generational search by flipping path conditions. The others are *bug-seeking queries* that attempt to synthesize inputs leading to specific kinds of bugs. Here `MallocArg` refers to a set of bug-seeking queries that attempt to force inputs to known memory allocation functions to be negative, yielding an implicit conversion to a large unsigned integer, or force the input to be small.

Experience Reporting to Developers. Our original strategy was to report all distinct bugs to developers and let them judge which ones to fix. The `mplayer` developers gave us feedback on this strategy. They wanted to focus on fixing the most serious bugs, so they preferred seeing reports only for out-of-bounds writes and double free errors. In contrast, they were not as interested in out-of-bound reads, even if the resulting read caused a segmentation fault. This helped us prioritize bugs for reporting.

7.3 Bug Statistics

Integer Bug-Seeking Queries Yield Bugs. Figure 6 reports the number of each type of query to the constraint solver over all test runs. For each type of query, we report the number of test files generated and the number of distinct bugs, as measured by our fuzzy stack hash. Some bugs may be revealed by multiple different kinds of queries, so there may be overlap between the bug counts in two different rows of Figure 6.

The table shows that our dynamic test generation methods for integer bugs succeed in finding bugs in our test programs. Furthermore, the queries for signed/unsigned bugs found the most distinct bugs out of all bug-seeking queries. This shows that our novel method for detecting signed/unsigned bugs (Section 5) is effective at finding bugs.

SmartFuzz Finds More Bugs Than zzuf, on mplayer. For `mplayer`, SmartFuzz generated 10,661 test cases over all test runs, while `zzuf` generated 11,297 test cases; SmartFuzz found 22 bugs while `zzuf` found 13. Therefore, in terms of number of bugs, SmartFuzz outperformed `zzuf` for testing `mplayer`. Another surprising result here is that SmartFuzz generated nearly as many test cases as `zzuf`, despite the additional overhead for

	SLOC	seedfile type and size	Branches	x86 instrs	IRStmts	asserts	queries
mplayer	723468	MP3 (159000 bytes)	20647045	159500373	810829992	1960	36
ffmpeg	304990	AVI (980002 bytes)	4147710	19539096	115036155	4778690	462346
exiv2	57080	JPG (22844 bytes)	809806	6185985	32460806	81450	1006
gzip	140036	TAR.GZ (14763 bytes)	24782	161118	880386	95960	13309
bzip	26095	TAR.BZ2 (618620 bytes)	107396936	746219573	4185066021	1787053	314914
ImageMagick	300896	PNG (25385 bytes)	98993374	478474232	2802603384	583	81

Figure 4: The size of our test programs. We report the source lines of code for each test program and the size of one of our seed files, as measured by David A. Wheeler's `sloccount`. Then we run the test program on that seed file and report the total number of branches, x86 instructions, Valgrind IR statements, STP assert statements, and STP query statements for that run. We ran symbolic execution for a maximum of 12 hours, which was sufficient for all programs except `mplayer`, which terminated during symbolic execution.

symbolic execution and constraint solving. This shows the effect of the generational search and the choice of memory model; we leverage a single expensive symbolic execution and fast solver queries to generate many test cases. At the same time, we note that `zzuf` found a serious InvalidWrite bug, while SmartFuzz did not.

A previous version of our infrastructure had problems with test cases that caused the target program to run forever, causing the search to stall. Therefore, we introduced a timeout, so that after 300 CPU seconds, the target program is killed. We manually examined the output of `memcheck` from all killed programs to determine whether such test cases represent errors. For `gzip` we discovered that SmartFuzz created six such test cases, which account for the two out-of-bounds read (InvalidRead) errors we report; `zzuf` did not find any hanging test cases for `gzip`. We found no other hanging test cases in our other test runs.

Different Bugs Found by SmartFuzz and `zzuf`. We ran the same target programs with the same seed files using `zzuf`. Figure 7 shows bugs found by each type of fuzzer. With respect to each tool, SmartFuzz found 37 total distinct bugs and `zzuf` found 59 distinct bugs. We found some overlap between bugs as well: 19 bugs were found by both fuzz testing tools, for a total of 77 distinct bugs. This shows that while there is overlap between the two tools, SmartFuzz finds bugs that `zzuf` does not and vice versa. Therefore, it makes sense to try both tools when testing software.

Note that we did not find any bugs for `bzip2` with either fuzzer, so neither tool was effective on this program. This shows that fuzzing is not always effective at finding bugs, especially with a program that has already seen attention for security vulnerabilities. We also note that SmartFuzz found InvalidRead errors in `gzip` while `zzuf` found no bugs in this program. Therefore `gzip` is a case where SmartFuzz's directed testing is able to trigger a bug, but purely random testing is not.

Block Coverage. We measured the number of basic blocks in the program visited by the execution of the seed file, then measured how many new basic blocks were visited during the test run. We discovered `zzuf` added a higher percentage of new blocks than SmartFuzz in 13 of the test runs, while SmartFuzz added a higher percentage of new blocks in 4 of the test runs (the SmartFuzz `convert-2` test run terminated prematurely.) Table 8 shows the initial basic blocks, the number of blocks added, and the percentage added for each fuzzer. We see that the effectiveness of SmartFuzz varies by program; for `convert` it is particularly effective, finding many more new basic blocks than `zzuf`.

Contrasting SmartFuzz and zzuf Performance. Despite the limitations of random testing, the blackbox fuzz testing tool `zzuf` found bugs in four out of our six test programs. In three of our test programs, `zzuf` found more bugs than SmartFuzz. Furthermore, `zzuf` found the most serious InvalidWrite errors, while SmartFuzz did not. These results seem surprising, because SmartFuzz exercises directed testing based on program behavior while `zzuf` is a purely blackbox tool. We would expect that SmartFuzz should find all the bugs found by `zzuf`, given an unbounded amount of time to run both test generation methods.

In practice, however, the time which we run the methods is limited, and so the question is which bugs are discovered first by each method. We have identified possible reasons for this behavior, based on examining the test cases and Valgrind errors generated by both tools[4]. We now list and briefly explain these reasons.

Header parsing errors. The errors we observed are often in code that parses the header of a file format. For example, we noticed bugs in functions of `mplayer` that parse MP3 headers. These errors can be triggered by simply placing "wrong" values in data fields of header files. As a result, these errors do not require complicated and unlikely predicates to be true before reaching buggy code, and so the errors can be reached without needing the full power of dynamic test generation. Similarly, code cov-

[4]We have placed representative test cases from each method at `http://www.metafuzz.com/example-testcases.tgz`

	mplayer		ffmpeg		exiv2		gzip		convert	
SyscallParam	4	3	2	3	0	0	0	0	0	0
UninitCondition	13	1	1	8	0	0	0	0	3	8
UninitValue	0	3	0	3	0	0	0	0	0	2
Overlap	0	0	0	1	0	0	0	0	0	0
Leak_DefinitelyLost	2	2	2	4	0	0	0	0	0	0
Leak_PossiblyLost	2	1	0	2	0	1	0	0	0	0
InvalidRead	1	2	0	4	4	6	2	0	1	1
InvalidWrite	0	1	0	3	0	0	0	0	0	0
Total	22	13	5	28	4	7	2	0	4	11
Cost per bug	$1.30	$2.16	$5.76	$1.03	$1.80	$1.03	$3.60	NA	$1.20	$0.65

Figure 7: The number of bugs, after fuzzy stack hashing, found by SmartFuzz (the number on the left in each column) and `zzuf` (the number on the right). We also report the cost per bug, assuming $0.10 per small compute-hour, $0.40 per large compute-hour, and 3 runs of 24 hours each per target for each tool.

erage may be affected by the style of program used for testing.

Difference in number of bytes changed. The two different methods change a vastly different number of bytes from the original seed file to create each new test case. SmartFuzz changes exactly those bytes that must change to force program execution down a single new path or to violate a single property. Below we show that in our programs there are only a small number of constraints on the input to solve, so a SmartFuzz test case differs from its seed file by only a small number of bytes. In contrast, `zzuf` changes a fixed fraction of the input bytes to random other bytes; in our experiments, we left this at the default value of 0.004.

The large number of changes in a single fuzzed test case means that for header parsing or other code that looks at small chunks of the file independently, a single `zzuf` test case will exercise many different code paths with fuzzed chunks of the input file. Each path which originates from parsing a fuzzed chunk of the input file is a potential bug. Furthermore, because Valgrind `memcheck` does not necessarily terminate the program's execution when a memory safety error occurs, one such file may yield multiple bugs and corresponding bug buckets.

In contrast, the SmartFuzz test cases usually change only one chunk and so explore only one "fuzzed" path through such code. Our code coverage metric, unfortunately, is not precise enough to capture this difference because we measured block coverage instead of path coverage. This means once a set of code blocks has been covered, a testing method receives no further credit for additional paths through those same code blocks.

Small number of generations reached by SmartFuzz. Our 24 hour experiments with SmartFuzz tended to reach a small number of generations. While previous work on whitebox fuzz testing shows that most bugs are found in the early generations [14], this feeds into the previous issue because the number of differences between the fuzzed file and the original file is proportional to the generation number of the file. We are exploring longer-running SmartFuzz experiments of a week or more to address this issue.

Loop behavior in SmartFuzz. Finally, SmartFuzz deals with loops by looking at the unrolled loop in the dynamic program trace, then attempting to generate test cases for each symbolic `if` statement in the unrolled loop. This strategy is not likely to create new test cases that cause the loop to execute for vastly more iterations than seen in the trace. By contrast, the `zzuf` case may get lucky by assigning a random and large value to a byte sequence in the file that controls the loop behavior. On the other hand, when we looked at the `gzip` bug found by SmartFuzz and not by `zzuf`, we discovered that it appears to be due to an infinite loop in the `inflate_dynamic` routine of `gzip`.

7.4 SmartFuzz Statistics

Integer Bug Queries Vary By Program. Table 5 shows the number of solver queries of each type for one of our 24-hour test runs. We see that the type of queries varies from one program to another. We also see that for `bzip2` and `mplayer`, queries generated by type inference for signed/unsigned errors account for a large fraction of all queries to the constraint solver. This results from our choice to eagerly generate new test cases early in the program; because there are many potential integer bugs in these two programs, our symbolic traces have many integer bug-seeking queries. Our design choice of using an independent tool such as `memcheck` to filter the resulting test cases means we can tolerate such a large number of queries because they require little human oversight.

Time Spent In Each Task Varies By Program. Figure 9 shows the percentage of time spent in symbolic execution, coverage, triage, and recording for each run of our experiment. We also report an "Other" category, which includes the time spent in the constraint solver. This shows us where we can obtain gains through further optimization. The amount of time spent in each task depends greatly on the seed file, as well as on the

Test run	Initial basic blocks		Blocks added by tests		Ratio of prior two columns	
	SmartFuzz	zzuf	SmartFuzz	zzuf	SmartFuzz	zzuf
mplayer-1	7819	7823	5509	326	70%	4%
mplayer-2	11375	11376	908	1395	7%	12%
mplayer-3	11093	11096	102	2472	0.9%	22%
ffmpeg-1	6470	6470	592	20036	9.14%	310%
ffmpeg-2	6427	6427	677	2210	10.53%	34.3%
ffmpeg-3	6112	611	97	538	1.58%	8.8%
convert-1	8028	8246	2187	20	27%	0.24%
convert-2	8040	8258	2392	6	29%	0.073%
convert-3	NA	10715	NA	1846	NA	17.2%
exiv2-1	9819	9816	2934	3560	29.9%	36.3%
exiv2-2	9811	9807	2783	3345	28.3%	34.1%
exiv2-3	9814	9810	2816	3561	28.7%	36.3%
gzip-1	2088	2088	252	334	12%	16%
gzip-2	2169	2169	259	275	11.9%	12.7%
gzip-3	2124	2124	266	316	12%	15%
bzip2-1	2779	2778	123	209	4.4%	7.5%
bzip2-2	2777	2778	125	237	4.5%	8.5%
bzip2-3	2823	2822	115	114	4.1%	4.0%

Figure 8: Coverage metrics: the initial number of basic blocks, before testing; the number of blocks added during testing; and the percentage of blocks added.

	Total	SymExec	Coverage	Triage	Record	Other
gzip-1	206522s	0.1%	0.06%	0.70%	17.6%	81.6%
gzip-2	208999s	0.81%	0.005%	0.70%	17.59%	80.89%
gzip-3	209128s	1.09%	0.0024%	0.68%	17.8%	80.4%
bzip2-1	208977s	0.28%	0.335%	1.47%	14.0%	83.915%
bzip2-2	208849s	0.185%	0.283%	1.25%	14.0%	84.32%
bzip2-3	162825s	25.55%	0.78%	31.09%	3.09%	39.5%
mplayer-1	131465s	14.2%	5.6%	22.95%	4.7%	52.57%
mplayer-2	131524s	15.65%	5.53%	22.95%	25.20%	30.66%
mplayer-3	49974s	77.31%	0.558%	1.467%	10.96%	9.7%
ffmpeg-1	73981s	2.565%	0.579%	4.67%	70.29%	21.89%
ffmpeg-2	131600s	36.138%	1.729%	9.75%	11.56%	40.8%
ffmpeg-3	24255s	96.31%	0.1278%	0.833%	0.878%	1.8429%
convert-1	14917s	70.17%	2.36%	24.13%	2.43%	0.91%
convert-2	97519s	66.91%	1.89%	28.14%	2.18%	0.89%
exiv2-1	49541s	3.62%	10.62%	71.29%	9.18%	5.28%
exiv2-2	69415s	3.85%	12.25%	65.64%	12.48%	5.78%
exiv2-3	154334s	1.15%	1.41%	3.50%	8.12%	85.81%

Figure 9: The percentage of time spent in each of the phases of SmartFuzz. The second column reports the total wall-clock time, in seconds, for the run; the remaining columns are a percentage of this total. The "other" column includes the time spent solving STP queries.

target program. For example, the first run of `mplayer`, which used a mp3 seedfile, spent 98.57% of total time in symbolic execution, and only 0.33% in coverage, and 0.72% in triage. In contrast, the second run of `mplayer`, which used a mp4 seedfile, spent only 14.77% of time in symbolic execution, but 10.23% of time in coverage, and 40.82% in triage. We see that the speed of symbolic execution and of triage is the major bottleneck for several of our test runs, however. This shows that future work should focus on improving these two areas.

7.5 Solver Statistics

Related Constraint Optimization Varies By Program. We measured the size of all queries to the constraint solver, both before and after applying the related constraint optimization described in Section 4. Figure 10 shows the average size for queries from each test program, taken over all queries in all test runs with that program. We see that while the optimization is effective in all cases, its average effectiveness varies greatly from one test program to another. This shows that different programs vary greatly in how many input bytes influence each query.

The Majority of Queries Are Fast. Figure 10 shows the empirical cumulative distribution function of STP solver times over all our test runs. For about 70% of the test cases, the solver takes at most one second. The maximum solver time was about 10.89 seconds. These results reflect our choice of memory model and the effectiveness of the related constraint optimization. Because of these, the queries to the solver consist only of operations over bitvectors (with no array constraints), and most of the sets of constraints sent to the solver are small, yielding fast solver performance.

8 Conclusion

We described new methods for finding integer bugs in dynamic test generation, and we implemented these methods in SmartFuzz, a new dynamic test generation tool. We then reported on our experiences building the web site `metafuzz.com` and using it to manage test case generation at scale. In particular, we found that SmartFuzz finds bugs not found by `zzuf` and vice versa, showing that a comprehensive testing strategy should use both

	average before	average after	ratio (before/after)
mplayer	292524	33092	8.84
ffmpeg	350846	83086	4.22
exiv2	81807	10696	7.65
gzip	348027	199336	1.75
bzip2	278980	162159	1.72
convert	2119	1057	2.00

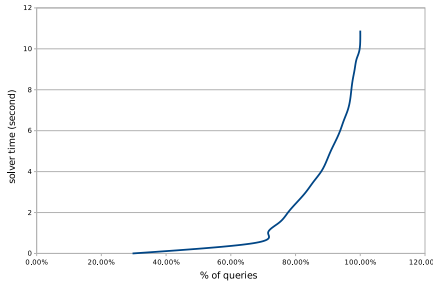

Figure 10: On the left, average query size before and after related constraint optimization for each test program. On the right, an empirical CDF of solver times.

white-box and black-box test generation tools.

Furthermore, we showed that our methods can find integer bugs without the false positives inherent to static analysis or runtime checking approaches, and we showed that our methods scale to commodity Linux media playing software. The Metafuzz web site is live, and we have released our code to allow others to use our work.

9 Acknowledgments

We thank Cristian Cadar, Daniel Dunbar, Dawson Engler, Patrice Godefoid, Michael Levin, and Paul Twohey for discussions about their respective systems and dynamic test generation. We thank Paul Twohey for helpful tips on the engineering of test machines. We thank Chris Karlof for reading a draft of our paper on short notice. We thank the SUPERB TRUST 2008 team for their work with Metafuzz and SmartFuzz during Summer 2008. We thank Li-Wen Hsu and Alex Fabrikant for their help with the metafuzz.com web site, and Sushant Shankar, Shiuan-Tzuo Shen, and Mark Winterrowd for their comments. We thank Erinn Clark, Charlie Miller, Prateek Saxena, Dawn Song, the Berkeley BitBlaze group, and the anonymous Oakland referees for feedback on earlier drafts of this work. This work was generously supported by funding from DARPA and by NSF grants CCF-0430585 and CCF-0424422.

References

[1] ASLANI, M., CHUNG, N., DOHERTY, J., STOCKMAN, N., AND QUACH, W. Comparison of blackbox and whitebox fuzzers in finding software bugs, November 2008. TRUST Retreat Presentation.

[2] BLEXIM. Basic integer overflows. *Phrack 0x0b* (2002).

[3] BLEXIM. Basic integer overflows. *Phrack 0x0b*, 0x3c (2002). `http://www.phrack.org/archives/60/p60-0x0a.txt`.

[4] BRUMLEY, D., CHIEH, T., JOHNSON, R., LIN, H., AND SONG, D. RICH : Automatically protecting against integer-based vulnerabilities. In *NDSS (Symp. on Network and Distributed System Security)* (2007).

[5] BRUMLEY, D., NEWSOME, J., SONG, D., WANG, H., AND JHA, S. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (2006).

[6] CADAR, C., DUNBAR, D., AND ENGLER, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of OSDI 2008* (2008).

[7] CADAR, C., AND ENGLER, D. EGT: Execution generated testing. In *SPIN* (2005).

[8] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. EXE: Automatically Generating Inputs of Death. In *ACM CCS* (2006).

[9] CHEN, K., AND WAGNER, D. Large-scale analysis of format string vulnerabilities in debian linux. In *PLAS - Programming Languages and Analysis for Security* (2007). `http://www.cs.berkeley.edu/~daw/papers/fmtstr-plas07.pdf`.

[10] CORPORATION, M. Vulnerability Type Distributions in CVE, May 2007. `http://cve.mitre.org/docs/vuln-trends/index.html`.

[11] DEMOTT, J. The evolving art of fuzzing. In *DEF CON 14* (2006). `http://www.appliedsec.com/files/The_Evolving_Art_of_Fuzzing.odp`.

[12] GANESH, V., AND DILL, D. STP: A decision procedure for bitvectors and arrays. CAV 2007, 2007. `http://theory.stanford.edu/~vganesh/stp.html`.

[13] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed Automated Random Testing. In *Proceedings of PLDI'2005 (ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation)* (Chicago, June 2005), pp. 213–223.

[14] GODEFROID, P., LEVIN, M., AND MOLNAR, D. Automated Whitebox Fuzz Testing. In *Proceedings of NDSS'2008 (Network and Distributed Systems Security)* (San Diego, February 2008). http://research.microsoft.com/users/pg/public_psfiles/ndss2008.pdf.

[15] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Active Property Checking. Tech. rep., Microsoft, 2007. MSR-TR-2007-91.

[16] HOCEVAR, S. zzuf, 2007. http://caca.zoy.org/wiki/zzuf.

[17] LANZI, A., MARTIGNONI, L., MONGA, M., AND PALEARI, R. A smart fuzzer for x86 executables. In *Software Engineering for Secure Systems, 2007. SESS '07: ICSE Workshops 2007* (2007). http://idea.sec.dico.unimi.it/~roberto/pubs/sess07.pdf.

[18] LARSON, E., AND AUSTIN, T. High Coverage Detection of Input-Related Security Faults. In *Proceedings of 12th USENIX Security Symposium* (Washington D.C., August 2003).

[19] LEBLANC, D. Safeint 3.0.11, 2008. http://www.codeplex.com/SafeInt.

[20] LMH. Month of kernel bugs, November 2006. http://projects.info-pull.com/mokb/.

[21] MICROSOFT CORPORATION. Prefast, 2008.

[22] MILLER, B. P., FREDRIKSEN, L., AND SO, B. An empirical study of the reliability of UNIX utilities. *Communications of the Association for Computing Machinery 33*, 12 (1990), 32–44.

[23] MOORE, H. Month of browser bugs, July 2006. http://browserfun.blogspot.com/.

[24] NETHERCOTE, N., AND SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI - Programming Language Design and Implementation* (2007).

[25] O'CALLAHAN, R. Chronicle-recorder, 2008. http://code.google.com/p/chronicle-recorder/.

[26] SEWARD, J., AND NETHERCOTE, N. 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.

[27] VUAGNOUX, M. Autodafe: An act of software torture. In *22nd Chaos Communications Congress, Berlin, Germany* (2005). autodafe.sourceforge.net.

[28] WANG, T., WEI, T., LIN, Z., AND ZOU, W. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Network Distributed Security Symposium (NDSS)* (2009).