The Past, Present, and Future of Cyberdyne

Peter Goodman and Artem Dinaburg | Trail of Bits

Cyberdyne—a distributed system that discovers vulnerabilities in third-party, off-the-shelf binary programs—competed in all rounds of DARPA's Cyber Grand Challenge. Since then, Cyberdyne has been successfully applied during commercial code audits. We describe its evolution and implementation as well as what it took to have it audit real applications.

The Trail of Bits cyber reasoning system, Cyberdyne, is an automated and distributed bug-finding system. It competed in all rounds of DARPA's Cyber Grand Challenge (CGC), first with the Trail of Bits team, and then as the bug-finding arm of one of the finalists. Since then, it has been successfully used to audit shipping software libraries and for commercial code audits.

Cyberdyne discovers and exploits memory access violations and information disclosure bugs. Like most CGC competitors, Cyberdyne applied two complementary techniques for finding these kinds of bugs. The first technique, fuzzing, repeatedly executes a program on inputs generated by mutating a common seed input. The second and more complex technique, symbolic execution, produces inputs that exercise all feasible program paths.

This article is divided into two separate but equally important halves. The first half of this article will describe Cyberdyne's unique features and approaches. Cyberdyne evolved over the course of the competition into a production quality bug-finding engine. Each component of Cyberdyne is a unique artifact, whose designs were motivated by observations and experimentation.

The second half of this article answers the big question that everyone had after the CGC: What's next? For Cyberdyne, the next step was auditing Linux programs for security vulnerabilities. Despite the deep differences between the OS used for the CGC (DECREE) and Linux (like the lack of files or threads), the vast majority of Cyberdyne could be reused to automatically identify bugs in real Linux applications. We will discuss how Cyberdyne performed the first paid automated security audit, and the challenges of automated security audits. We conclude this article with a discussion about the future of automated bug-finding systems and how automated security assessments will improve software quality and security.

Building Cyberdyne

Architecture

Cyberdyne is a distributed system that tries to prove that a target program has memory access violation bugs or information disclosure bugs. Instead of attacking this problem head-on, Cyberdyne implements a genetic algorithm to produce inputs (genes) that maximize the amount of code executed (fitness function) by the target program. If these inputs crash the target program, Cyberdyne can show the program has memory safety bugs.

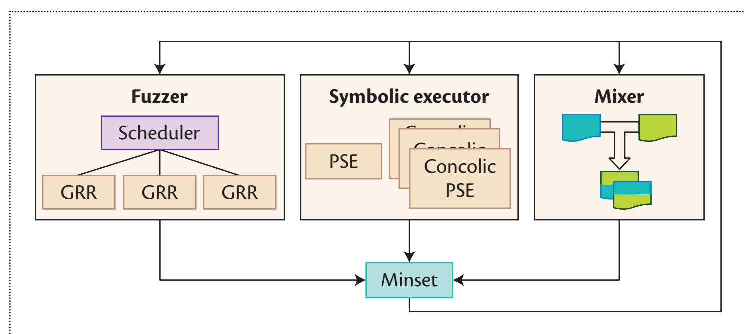

Figure 1. The architectural diagram of a Cyberdyne node.

Nodes in a Cyberdyne cluster mostly operate in isolation, with each node focusing on finding vulnerabilities in one or more target programs. When there are more nodes than target programs, nodes cooperate to share the workload. Cooperating nodes self-configure to use different metrics for bug-finding progress in an attempt to inspect the most target program states.

As shown in Figure 1, each node in a Cyberdyne cluster is composed of four main services:

- the fuzzer,
- the symbolic executor,
- the mixer, and
- the minset.

These services share data and communicate with one another over a shared Redis server. The first two services, the fuzzer (GRR; https://github.com/trailofbits/grr) and symbolic executor (PySymEmu running in symbolic and concolic modes; https://github.com/trailofbits/manticore) are bug-discovery tools. The fuzzer performs input mutation; the symbolic executor synthesizes new inputs. The mixer service performs input crossover, and the minset service implements the fitness function. Fitness is evaluated by how much unique code coverage an input contributes to the set of coverage induced by all generated inputs.

The following sections describe each Cyberdyne service in turn.

Fuzzer

Fuzzing is a software assurance methodology that uses input generation and concrete execution to discover security faults. Fuzzing is widely accepted in the software security industry and is used by large companies such as Microsoft, Google, and Adobe. Any fault produced by a fuzzer represents a real fault in the program. However, the absence of identified faults does not imply the program is bug-free.

Fuzzers work by feeding randomized inputs into a program and determining whether these inputs cause the program to crash or to execute new code. Modern fuzzers instrument the target program and modify inputs based on feedback from prior executions.

Cyberdyne's fuzzer service has two components: the scheduler and GRR. The scheduler orchestrates GRR and decides what, when, and how to fuzz. GRR feeds and mutates inputs to the program, and instruments the program to determine when an input causes a crash.

Scheduler. By default, the scheduler devotes an equal share of CPU resources to fuzzing each program. When a vulnerability in a program is discovered, the node responsible broadcasts this fact to others, thereby reducing CPU resources dedicated to fuzzing that program. This was a prudent choice for the competition, where the goal was to exploit as many programs as possible.

The scheduler operates on the list of inputs supplied by the minset service, which is described later. By focusing on a limited set of inputs, the fuzzer makes efficient use of limited CPU resources. A positive side-effect of using the minset to select fuzzing inputs is that the fuzzer cooperates with other input sources (for instance, the symbolic executor). Inputs in the minset could, and did, have a mixed ancestry: an input may first be created via symbolic execution, then modified several times via fuzzing before crashing the target. One way to visualize cooperation between tools is as a kind of hill-climbing. The fuzzer mutates inputs that are highest on the hill, hoping that the mutation will yield new code coverage even higher up.

The inputs themselves are prioritized for fuzzing by recency. Inputs most recently added to the minset are allotted more CPU resources than those added long ago. This fits with the hill-climbing theme; recent inputs more likely exercise deeper program states (higher on the hill), by virtue of being derived from inputs produced earlier (lower on the hill).

Separating scheduling from mutation allows us to make the scheduler smart and the mutator extremely fast. While the scheduler selects what to mutate, GRR performs the mutation.

GRR. GRR is an emulator with built-in support for mutating inputs and instrumenting program execution. GRR was designed to maximize throughput: a single GRR process can cycle between mutating an input and performing an instrumented execution of that mutated input. In the CGC, a single GRR process executing for one hour would typically perform one million input–execute cycles.

Dynamic binary translation. GRR is a 64-bit x86 program that emulates 32-bit x86 instructions using

dynamic binary translation (DBT). It dynamically decodes a sequence of 32-bit x86 instructions (called a basic block), modifies the decoded instructions (perhaps adding in new ones), and encodes and later executes the new instruction sequence. When executed, the new instruction sequence performs the same operations as the original sequence, but with additional instrumentation (for example, recording what code is executed, intercepting memory accesses).

As a 64-bit program, GRR can use more hardware registers and memory than the original program. This setup simplified translation from 32-bit to 64-bit machine code. The translated machine code could use an additional eight registers guaranteed not to interfere with any registers used by the original machine code. Each of these eight "newly available" registers were given specific meanings in the translated code. For example, the translator rewrote all memory-accessing instructions to use a special "MEMORY" base register that pointed at the base of the emulated 32-bit address space.

Code caching. Existing DBTs (for instance, Intel PIN, DynamoRIO) retranslate the same program every execution. They specialize in translating long-running programs, where the translation cost is amortized over time, and "hot" code is reorganized to improve performance. DBTs avoid retranslating the same code by storing translations in an in-memory cache, and indexing that cache with a lookup table. Fuzzing campaigns like the CGC and code audits require billions of program executions. This means that all code in a target program, even in short-running programs, is hot. This realization motivated GRR's code cache and index persistence feature. Independent executions of GRR need not retranslate the same code.

A unique feature of GRR's persisted cache index is that it permits caching of self-modified or just-in-time compiled code.

GRR's cached translations can also be reused for different analysis purposes. For example, the same cached translations can be used for recording code coverage and memory access interception. This flexibility is achieved using a combination of specific meanings for each additional 64-bit register and a "stub function" mechanism for instrumentation callbacks.

A unique feature of GRR's persisted cache index is that it permits caching of self-modified or just-in-time compiled code. Typical DBT cache indices map the virtual addresses of original instructions to those of translated instruction. GRR's index maps original, 32-bit instruction addresses and a code "version number" to the offset of the translated code within the persisted cache file. The version number is a Merkle hash of the contents of executable memory at the time that the instruction was translated. Modifying the contents of an executable page in memory invalidates its hash, thereby triggering retranslation of any code on that page when it's next executed.

Snapshotting. GRR is actually two programs: the first program takes "snapshots," and the second program emulates executions, using the snapshots as a template for the program's initial state. Snapshotting was motivated by the observation that programs typically execute deterministic setup code prior to receiving input. Snapshotting execution prior to reading external input allows GRR to skip setup code and to ensure such code does not contribute to code coverage.

OS and I/O. GRR emulates programs that interact with DECREE, a custom, Linux-like operating system developed by DARPA for the CGC. GRR is a single-threaded program; however, it can emulate concurrent target programs communicating via sockets. For example, GRR can emulate a server, a client program, or both at the same time. GRR implements a simple round-robin process scheduler, swapping process contexts between system calls.

GRR emulates all I/O in memory and can perform millions of independent input mutations and execution emulations during a single run of the GRR process. This setup makes GRR entirely CPU-bound.

How GRR fuzzes. A typical GRR fuzzing run begins by executing a target program on an initial input file and recording how the target program consumes the input. The recording of all the input entering the program, and how the input arrives, serves as the basis for GRR's input mutation.

GRR mutates input recordings via three different granularities. The smallest granularity, system call, applies a mutation operator to the input bytes of one or more consecutive read system calls. For interactive programs, this often represents byte-granularity mutation. For structured or layered input formats, this can enable mutation of the embedded data. The second granularity, line granularity, compresses the system call recording, combining uninterrupted sequences of read system calls into logical "lines" of input. Other I/O system calls represent interruption points. For some interactive programs, this enables mutation of actual user-supplied "answers" to input prompts. The

last granularity, file granularity, behaves just like other fuzzers: a mutation operator is applied to the initial input as a whole.

The fundamental mutation operators (for example, flipping a bit in each input byte) are implemented as transformations within GRR. For more complex mutations, GRR can embed external mutators. By default, GRR uses Radamsa (an open source input mutation engine; https://github.com/aoh/radamsa) when mutating longer inputs.

GRR's recording and replay can also be used to explore program inputs generated by other sources. One of the key input sources, described in the next section, is the symbolic executor, which is a crucial source of the raw input data used for mutation.

Symbolic Executor

PySymEmu (PSE) is a custom symbolic execution engine written in Python and was originally developed prior to Cyberdyne. We chose PSE because it was easy to understand, extend, and integrate into Cyberdyne. PSE is composed of three main parts: a symbolic CPU, a memory model, and an operating system model.

PSE operates directly on x86 machine code and runs by iteratively decoding and emulating x86 machine instructions. Because PSE reads each instruction from emulated memory, it can handle edge cases like self-modifying code that can stymie other symbolic executors. The PSE memory model is specifically suited for analyzing binaries: PSE divides memory into pages, which can contain a mix of concrete and symbolic bytes, and can be addressed via both concrete and symbolic memory addresses.

PSE's operating system model supports symbolic data as arguments to system calls. In such cases, PSE will fork the analysis to explore all possible concrete states. PSE also implements basic support for multiprocessing.

As with any symbolic execution tool, PSE has to choose from many possible program states to analyze. PSE totally orders all states according to a metric based on the number of input bytes read, output bytes written, and total and unique instructions executed. The same metric is applied to all programs. As a hedge against this metric being a poor choice for a particular program, PSE will randomly choose between the top and bottom five ranked states. Fundamentally, this state-ordering metric provides no deep insights: it was experimentally chosen by trial and error.

What made PSE effective in Cyberdyne was its ability to use GRR program snapshots to simulate concolic operation as well as its approach to producing inputs. In the former case, PSE could sidestep the typical state explosion scalability issue by "jumping in" and beginning full symbolic execution deep within some program state. This also enabled cooperation between PSE and other input producers (for instance, GRR). In the latter case, PSE produced inputs at every symbolic fork. That is, the vast majority of inputs produced by PSE were produced before PSE observed the target program's termination. Frequent and rapid input production enabled deeper program exploration by the fuzzer.

KLEE. Early versions of Cyberdyne also integrated KLEE (https://klee.github.io), another open source symbolic executor. KLEE operates on LLVM IR (LLVM Intermediate Representation), a program representation used by the clang compiler. Typically, LLVM IR is produced by compiling C/C++ source code with clang. In the CGC, source code for target programs was not available. We overcame this challenge by using McSema (https://github.com/trailofbits/mcsema) to convert x86 binaries into LLVM IR.

Cyberdyne's KLEE bug-finding service was eventually deprecated because KLEE was challenging to understand and maintain and because its symbolic emulation was a "leaky abstraction." In the latter case, KLEE provides no isolation between its runtime (for example, memory and code used by its OS emulation functions) and that of the program being symbolically executed. This led to false positives (KLEE claiming inputs crashed the program) and false negatives (KLEE missing actual crashes).

Mixer

The mixer performs the crossover operation of a generic algorithm: it takes existing inputs with high code coverage and uses various heuristics to merge these inputs into a new candidate input. In most systems, the input-mixing stage occurs as a part of whole-file mutation prior to fuzzing, but because GRR fuzzes programs via DBT, Cyberdyne can provide more granular mixing. For example, the mixer can take two program traces and interleave an input system call from each trace to produce a new input. Or it can operate at the multiple-call or whole-file level, splicing two inputs in various ways.

While it sounds simple, the mixer was surprisingly effective. For example, imagine a program where you enter your name and then solve a maze. If you solve the maze, an overly long name triggers a bug at the high score screen. Using the mixer, a series of inputs that solves the maze would quickly propagate to other high-coverage inputs, like one that sets an overly long name. In effect, the "genes" of high-coverage inputs would mix together to form child inputs that result in higher code coverage than the parents'.

Minset

The minset service implements the fitness function of Cyberdyne's genetic algorithm. Fitness is measured

using code coverage: if an input induces the execution of previously unexecuted code, then that input is scored as interesting and is eventually fed to the other services. The goal of the minset service is to select a minimum set of program inputs that maximize code coverage.

Coverage-guided bug-finding systems have become commonplace in recent years. American Fuzzy Lop (AFL; http://lcamtuf.coredump.cx/afl), libFuzzer (http://llvm.org/docs/LibFuzzer.html), and Sanitizer-Coverage are all production-quality coverage-guided fuzzers. Some of these tools were employed by other teams in the CGC. For example, the Shellphish team used AFL directly, and the Codejitsu team used a modified version of AFL (https://github.com/mboehme/aflfast) that employed a new algorithm for prioritizing which inputs to fuzz.

Code coverage. There are many ways of measuring how much of a program is executed given a specific input. Cyberdyne uses an extended form of branch coverage as its code coverage metric. Before executing a branch instruction, Cyberdyne records a three-address tuple:

- the most recently executed branch instruction address,
- the about-to-execute branch instruction address, and
- the destination instruction address of the about-to-execute branch.

> **After the CGC, the question was "what's next?" For us, the answer was retargeting the technology to audit the deployed programs and libraries that silently power the computing infrastructure.**

The most recently executed branch instruction can be arbitrarily far back in time, adding more sensitivity to this coverage metric.

Cyberdyne also counts indirect control flow instructions as branches (for instance, vtable-based method calls, jump tables implementing `switch` statements, and function returns). For example, function return instructions redirect control flow by jumping to an instruction address stored on the stack. Attackers can utilize stack-based buffer overflows to corrupt the stored return address and take control of a program's execution. Treating return instructions as branches allows Cyberdyne to detect return address changes as new coverage.

Lossy or lossless? What metrics count toward code coverage are just as important as how those metrics are recorded. A precise, or lossless, coverage recording adds any input that induces the execution of a previously unseen branch tuple into the minset. A lossy coverage recording has the potential to treat some inputs as not inducing new coverage even when it should.

In the CGC, we observed that importance of precision changed over time. The CGC was a competition, and points were scored when bugs were found, even if those bugs could not be converted into vulnerabilities. This incentivized the discovery of low-hanging fruit, that is, superficial bugs that manifest early in a program's execution, before the harder-to-find bugs are reached. Our experiments showed that most superficial bugs could be discovered within the first 30 minutes of fuzzing a target program, but only if the metric, or recording thereof, was extremely lossy. Hard-to-detect bugs, however, required precise coverage information recording.

We implemented adjustable precision control by taking inspiration from probabilistic data structures like Bloom filters. Cyberdyne recorded coverage by hashing each branch tuple and using that hash as an index into a file-backed bitmap. When a branch is executed, the instrumentation sets the bit associated with the hashed tuple. Low-precision recording was achieved by using a small bitmap file, or a hash function with a poor distribution, whereas high precision was achieved by using larger bitmap files and better hash functions.

Why did precision matter? The reason is a mix of resource allocation, scheduling, and brute-force effort. Shallow bugs can usually be discovered by brute force, that is, letting the fuzzer run its course of mutators given an input seed. Cyberdyne's fuzzer scheduler gives priority to "new" inputs and prioritized deterministic mutators (for example, flip the first bit of every byte) above nondeterministic ones (for instance, Radamsa). Starting a campaign with low-precision coverage recording helped keep the minset small, thereby bringing to bear more CPU resources per input in the minset.

Deployment

We automated the provisioning and deployment of a Cyberdyne system. A multinode Cyberdyne system can be deployed on private or public clouds via a single command line. We have successfully run Cyberdyne on hardware ranging from a developer's laptop to hundreds of extra-large Amazon EC2 instances. This flexibility allows us to use Cyberdyne for something as large as the Cyber Grand Challenge or for something as small as a single-application code audit.

Using Cyberdyne

First, we effectively used Cyberdyne during the CGC. Cyberdyne identified the second-most amount of bugs

in the CGC's qualification challenge binaries. A faster, more accurate, and more efficient version of Cyberdyne was a key component of team Deep Red's bug-finding operations in the final event.

After the CGC, the question was "what's next?" For us, the answer was retargeting the technology to audit the deployed programs and libraries that silently power the computing infrastructure we take for granted.

We began an effort to use Cyberdyne to automatically identify bugs in Linux applications. We knew that Cyberdyne could find bugs in "real" software—the challenge binaries are most certainly real.[1] They exhibit complex behavior and vulnerabilities unknown to the authors. However, Cyberdyne was designed to find bugs only in software written for DECREE, an operating system purpose-built for the CGC. While DECREE is Linux based, it is very different from Linux.

Using Cyberdyne on Linux Applications

There are substantial design differences between DECREE and Linux. DECREE was designed to remove unnecessary complexity so competitors could focus on the science of program analysis instead of modeling quirks of operating system abstractions. Although DECREE is based on the Linux kernel, operating system features such as threads, sockets, files, and signals are not accessible to DECREE applications. Linux programs expect all these features to work, and more.

Rather than making the effort to port Cyberdyne to Linux, we realized that we could audit real programs quickly if we ported them to run in DECREE. The porting process requires little to no modification of the original program source. Most of the effort focused on modifying how the programs were built and simulating Linux functionality in DECREE.

Simulating Linux with DECREE. First, we created a libc implementation using only DECREE functionality. The core of libc was built from parts of open source libc implementations and from libc portions implemented in the challenge binaries. Where necessary, such as for file, thread, and process operations, we mocked the functionality. Most mock system calls either return a successful error code or error not implemented, but a few, such as `open` and `time`, required more complex modeling. File operations on key files such as `stdin`, `stderr`, and `stdout` must be accurately modeled to get input into the program. Time retrieval operations shouldn't make time go backward and should report a time close to the present day. Resource limits for `rlimit` should be realistic, as should `stat` results for files that are known to exist on a real Linux system.

Building programs to LLVM bitcode. DECREE programs are built using clang. Porting a Linux application to DECREE is a four-step process. First, we obtain the source code for the application and all of its dependencies. Second, we compile all relevant source files to LLVM bitcode modules. Third, we link all modules together into a single, unified LLVM bitcode module. Finally, we link this aggregate module to our custom libc implementation. The result is a single file that represents the program and all dependencies, down to the system call layer.

Emitting DECREE executables. Combining all dependencies in a single LLVM bitcode module enables substantial optimization and analysis opportunities. A typical program will use only a fraction of libc and dependent library functionality. Normal libraries are a package deal: there is no way to import only some functionality. Because our new build system merges dependencies at the LLVM bitcode layer, we can use LLVM's optimization passes to eliminate unused functionality and to flatten layers of indirection. Less code in the program means less code to analyze. Because Cyberdyne operates on binaries, we can gain more analysis advantages by emitting only instructions that are easy to reason about, especially for our binary symbolic executor PSE. For instance, we can avoid creating a binary with legacy FPU instructions or complex vector instructions like AVX.

Limitations. Our approach to porting applications to DECREE has limitations. First, this approach requires source code. While access to source code is not a problem for most Linux applications or code-auditing engagements, it removes certain classes of programs from analysis. Second, some applications will never be portable to DECREE: they may require graphical interaction, shared memory, or other features that can't be duplicated in DECREE. That shouldn't stop us from porting those components we'd most want to analyze—encoders and decoders, compression libraries, image processing libraries, parsers, and so on.

Next, let's explore how we applied Cyberdyne to perform the first paid automated security audit in history.

The First Paid Automated Security Audit

In August 2016, Cyberdyne audited zlib (https://github.com/madler/zlib) for the Mozilla Secure Open Source (SOS) Fund. To our knowledge, this is the first instance of a paid, automated security audit. Zlib is an open source compression library that is used in virtually every software package that requires compression or decompression. This very article was created and printed with multiple software packages that use zlib. Online readers

are also using zlib—the PDF viewer or web browser you are using relies on zlib for compression and decompression functionality.

Zlib has a relatively small code base that hides a lot of complexity. First, the code that runs on the machine may not exactly match the source, due to compiler optimizations. Some bugs may only occur occasionally due to use of undefined behavior. Others may be triggered only under extremely exceptional conditions. In a well-inspected code base such as zlib, the only bugs left might be too subtle for a human to find during a typical engagement.

Mozilla is a nonprofit and houses a variety of projects beneficial to the public. Our automated audit of zlib was thousands of dollars cheaper than an equivalent human-powered audit and provided measureable code coverage and generated inputs. The money saved can be put to good use supporting other Mozilla projects, and the generated artifacts can be easily reused for future automated or manual audits.

For this automated assessment, we paired Cyberdyne with TrustInSoft's verification software (https://trust-in-soft.com) to identify memory corruption vulnerabilities, create inputs that stress varying program paths, and identify code that may lead to bugs in the future.

Table 1. Code coverage in zlib.

File	Line coverage (%)	Branch coverage (%)
adler32.c	67.20	61.80
compress.c	90.50	62.50
crc32.c	29.10	32.60
deflate.c	44.00	30.70
gzclose.c	80.00	75.00
gzlib.c	37.80	24.60
gzread.c	50.70	38.60
gzwrite.c	40.50	24.70
infback.c	0.00	0.00
inffast.c	83.80	75.70
inflate.c	75.30	65.90
inftrees.c	98.30	93.70
trees.c	93.60	89.40
uncompr.c	100.00	71.40
zutil.c	38.50	50.00

Audit methodology. During the assessment, we focused on typical zlib usage and code related to compression and decompression functionality. Unrelated features were not audited, unless they were called by core compression or decompression routines.

Zlib is written in C and is designed to build for an extremely wide variety of platforms and compilers. Some code is built only for certain platforms (for instance, only big endian or only little endian). For the audit, we built zlib version 1.2.8 (the latest available at the time) using the clang compiler targeting the 32-bit Intel x86 instruction set.

Audit results. Cyberdyne is especially tuned for identifying memory safety violations (for example, buffer overflows, use-after-free errors, stack overflows, and heap overflows). Using Cyberdyne, we were unable to identify memory safety issues with the `compress`, `uncompress`, `gzread`, and `gzwrite` functions in zlib. We concluded that the assessed code was highly unlikely to harbor these types of bugs. The TrustInSoft analyzer identified uses of undefined behavior; the full report describes the details of these potential vulnerabilities.[2]

The full line and branch coverage results for the core zlib source files are shown in Table 1 (reproduced from the audit report). Cyberdyne automatically generated inputs to gather this coverage, given a program to exercise the correct functionality and minimal seed inputs to speed up the input synthesis process. Very high coverage for the location of previous zlib vulnerabilities, in the Huffman tree code (inftrees.c: 98.3 percent line coverage, 93.7 percent branch coverage, trees.c: 93.6 percent line coverage, 89.4 percent branch coverage), was a very welcome sign. The lone outlier was the file infback.c, which had 0 percent coverage. Its functionality was never invoked. Infback.c is an alternative to inflate.c and is only used when callback style I/O is preferred.

This coverage is a result of invoking the compression-related code both directly and via gzip functionality. This compares very favorably to the handcrafted unit tests that come with zlib, which generate 100 percent coverage for infback.c, inffast.c, and inftrees.c; 98.6 percent coverage for inflate.c; and almost zero coverage for anything else.

Cyberdyne's automated audit revealed no new memory safety violations in zlib. Far from a disappointment, the null result was expected. Zlib has been audited by humans and battle-tested by virtue of being deployed on almost every Internet-connected device. To date,

there have been no reported vulnerabilities in the tested version of zlib.

The audit was a learning experience for us on the benefits, challenges, and limitations of automated code audits. While the actual auditing is fully automated, the kind of library software Cyberdyne is best at analyzing requires a level of upfront manual effort.

Challenges of Automated Security Audits

Cyberdyne automates the process of generating program inputs, exercising new code paths, and identifying bugs. What Cyberdyne does not automate is porting Linux applications to DECREE or writing programs to exercise library functionality. Porting and exercising functionality still require careful analysis of how the software under test works and initial manual effort.

Building the target software. The biggest challenges in porting software to DECREE are identifying all build dependencies and modifying all the build and configuration systems to emit code for 32-bit x86 processors, disabling potentially problematic features (for example, handwritten vectorization, threading support, and so on), and using a customized version of clang that emits whole-program bitcode. Even for projects that build with GNU Autotools, simply changing the compiler is rarely sufficient. Some software won't build with clang. Other software requires custom configuration options or has a customized build step. Most build configurations are just different enough from one another that some human intervention in the initial build process is still required.

"Automation will democratize access to security: low-cost, high-coverage testing will finally be within reach of individual developers and open source projects."

Exercising program functionality. Exercising program functionality is the second big challenge of automated auditing. End-user applications have a single entry point (that is, `main`) but process several command line flags that affect the program. Shared libraries export a range of functionality via different exported symbols. How program or library functionality should be used is not defined at a machine level, but via documentation (which is sometimes lacking).

To illustrate, let's use zlib as a motivating example. The usual entry points into zlib are `compress` and `uncompress`, but zlib also offers gzip wrappers around the standard library file functions (for example, `gzopen`, `gzread`, `gzwrite` and so on), checksumming functionality, and more. To test the whole library, a program or multiple programs must be written to invoke these functions. In Cyberdyne parlance, the small programs that feed input to functionality under test are called drivers. Developing drivers to exercise all program functionality is a part of the initial manual test setup process required to use Cyberdyne to audit Linux software.

Driver development ranges from the trivial to the complex, depending on how many entry points the underlying software has. Applications that accept file inputs typically need a single trivial driver (for example, call the main function with `stdin` as the input file). Shared libraries require many carefully written drivers that demand a thorough understanding of the software under test.

Depending on the software under test, other factors may also come into play during driver development: How do you set bounds on inputs to software that accepts unbounded input lengths? How do you best mock system configuration files read by the software? What are realistic values for mocked resource limits?

Even though it sounds like creating drivers is a daunting task, automation comes to the rescue once again. Because the actual auditing is fully automated, driver development can be iterative. An initial driver just needs to feed input to some part of the software. That driver will then start the initial audit. As the initial audit runs, any newly developed drivers can be added to Cyberdyne to increase the breadth of tested code.

The Future of Automated Code Audits

Despite the challenges, automated code audits are the future of security testing. Human auditors are rare and expensive and simply cannot keep up with the volume of newly written code. Automation will democratize access to security: low-cost, high-coverage testing will finally be within reach of individual developers and open source projects. The revolution in security testing will dramatically improve the security posture of the Internet's core infrastructure.

The automated auditing revolution has already begun. Projects like Google's oss-fuzz project and Coverity Scan already enable continuous fuzzing and static analysis checking of open source applications. We envision a future where services like Cyberdyne integrate with common continuous integration and code-hosting environments to deliver quality, low-cost security audits. Initially, automated audits will be a complement to unit and integration tests—something that good software developers use to ensure they ship quality code.

As automated audit technologies mature, objective security metrics will slowly become meaningful and change the economics of software security for the better. Imagine if software came with verifiable guarantees of "audited with Cyberdyne to 99.99 percent branch coverage." The label wouldn't guarantee bug-free operation, because exploitable vulnerabilities may still exist in edge cases, but it would be meaningful way to compare software products. Once objective security measures influence software purchasing decisions, there will finally be an economic reward for writing software securely. The economic incentives will create a virtuous cycle of better analysis tools, higher-quality software, and a more secure world for all.

There are no magic numbers that make Cyberdyne find bugs. There are reasons why even just one single-threaded GRR process can perform a million mutate, execute, and code coverage measurement cycles per hour. The decisions that we made were guided by a performance-focused mindset, and backed up with measurements. Our approach is general and broadly applicable to other bug-finding systems. The approach provides a framework for tackling issues when scaling vulnerability discovery, for instance, sidestepping state explosion in symbolic executors.

> **As automated tools improve, we believe they can finally upend the economics of software development to reward safety instead of features by providing objective, comparable security metrics.**

The tools and concepts developed for the Cyber Grand Challenge can be used on practical software. Cyberdyne demonstrated this by performing the first paid automated security audit for the Mozilla SOS fund. The automated audit of zlib delivered higher confidence at a lower cost and faster timeframe than was possible with qualified human code auditors.

Automation represents a shift in the way that software security audits can be performed. It's a tremendous step toward securing the Internet's core infrastructure. Automation can deliver continuous, low-cost, and effective security analysis. Other projects have already adopted this model: Google's oss-fuzz project enables continuous fuzzing of open source applications. We envision a similar future for Cyberdyne: a service that works with popular code-hosting and continuous integration environments and delivers quality, low-cost, continuous security audits. As automated tools improve, we believe they can finally upend the economics of software development to reward safety instead of features by providing objective, comparable security metrics across disparate software packages. ■

References

1. "Your Tool Works Better Than Mine? Prove It," Trail of Bits blog, 1 Aug. 2016; https://blog.trailofbits.com/2016/08/01/your-tool-works-better-than-mine-prove-it.
2. A. Dinaburg and P. Cuoq, *Zlib: Automated Security Assessment*, Trail of Bits, 30 Sept. 2016; https://github.com/trailofbits/public-reports/blob/master/Zlib-AutomatedSecurityAssessment.pdf.

Peter Goodman is a senior security engineer at Trail of Bits. He is an expert at designing and implementing binary translation and instrumentation systems. He holds an MS from the University of Toronto, where his research focus was operating system kernel instrumentation. In a past life, he was a competitive ski racer. Contact at peter@trailofbits.com.

Artem Dinaburg is a principal security engineer at Trail of Bits. His current interests include automated program analysis and creating program analysis tools usable by software developers. He holds a bachelor's in computer science from Penn State and a master's in computer science from Georgia Tech. He has previously spoken at academic and industry computer security conferences such as ACM CCS, INFILTRATE, Blackhat, and ReCON. Contact at artem@trailofbits.com.