MINESTRONE: Testing the SOUP

Azzedine Benameur
Symantec Research Labs
Herndon, VA
azzedine_benameur@symantec.com

Nathan S. Evans
Symantec Research Labs
Herndon, VA
nathan_evans@symantec.com

Matthew C. Elder
Symantec Research Labs
Herndon, VA
matthew_elder@symantec.com

***Abstract*—Software development using type-unsafe languages (e.g., C and C++) is a challenging task for several reasons, security being one of the most important. Ensuring that a piece of code is bug or vulnerability free is one of the most critical aspects of software engineering. While most software development life cycle processes address security early on in the requirement analysis phase and refine it during testing, it is not always sufficient. Therefore the use of commercial security tools has been widely adopted by the software industry to help identify vulnerabilities, but they often have a high false-positive rate and have limited effectiveness. In this paper we present MINESTRONE, a novel architecture that integrates static analysis, dynamic confinement, and code diversification to identify, mitigate, and contain a broad class of software vulnerabilities in Software Of Uncertain Provenance (SOUP). MINESTRONE has been tested against an extensive test suite and showed promising results. MINESTRONE showed an improvement of 34.6% over the state-of-the art for memory corruption bugs that are commonly exploited.**

I. Introduction

Shipping bug or vulnerability free software is a major concern for software engineering teams. Therefore efforts have been made to address security concerns early on and make it part of the software development life cycle [1]. While this reduces the number of bugs, it is not sufficient and software vulnerabilities are still growing [2]. The software industry relies on various products [3], [4] to detect vulnerabilities early on in type-unsafe languages such as C and C++. These products employ static analysis, which produces a large number of false positives that must be individually resolved by a domain expert, a time consuming process.

MINESTRONE [5] is a novel architecture that addresses software vulnerabilities for C/C++ languages by combining static and dynamic analysis, confinement, and code diversification. MINESTRONE is part of the IARPA STONESOUP program [6], addressing vulnerabilities in Software Of Uncertain Provenance (SOUP). The goal of STONESOUP (Securely Taking On New Executable Software Of Uncertain Provenance) is to develop "comprehensive, automated techniques that allow end users to securely execute software without basing risk mitigations on characteristics of provenance". In reality, given today's software development practices, the notion of provenance is indeterminate: it is impossible to know who developed a piece of software, where, and/or how. For example, companies outsource and offshore product development, and products incorporate open source components that might have been developed by any number of people in any number of locations. Instead, the STONESOUP program seeks to establish confidence in software based on the properties of the software itself, by examining it directly, independent of its provenance.

Existing vulnerability analysis tools address the problem of whether or not a piece of software is safe to run by examining it directly, but current tools – primarily based on static analysis – are difficult to use without significant expertise and produce a great deal of output, including a significant proportion of false positives. Furthermore, prior to the creation of STONESOUP, the NSA conducted a study of the state-of-the-art tools in source code analysis and found that each tool incurred a high percentage of false negatives [7]. For these reasons, the STONESOUP program seeks to detect and mitigate vulnerabilities in software via defense in depth, a combination of complementary techniques, with the goal that software can be first analyzed and then executed safely, contained and possibly transformed via diversification, by non-experts.

MINESTRONE is versatile architecture that incorporates a number of component technologies using a combination of analysis, confinement, and diversification approaches. MINESTRONE leverages confinement to mitigate potential vulnerabilities and can be deployed to transparently protect running applications using dynamic security instrumentation [8]. In this paper, we present our results and experiences conducting an evaluation of the MINESTRONE system and component technologies using a test suite of programs containing memory error (e.g., buffer overflow) and null pointer vulnerabilities developed by an independent test and evaluation team within the STONESOUP program.

The remainder of the paper is organized as follows. First, we present related work. Next, we detail our methodology and experimental setup in which we test MINESTRONE against a test suite containing memory error and null pointer vulnerabilities. Then we present the results of our experimentation with these test suites. Finally, we share our lessons learned and conclusions.

II. Related Work

There are a number of existing commercial tools for finding vulnerabilities in source code. In 2009, the NSA conducted a comparison of some of these tools [7] in order to evaluate the capabilities of the state of the art, in terms of both the breadth and depth of coverage. The tools evaluated in that study for finding vulnerabilities in C/C++ code were Coverity Prevent, Fortify SCA, GrammaTech CodeSonar, Klocwork Insight, and Ounce Labs Ounce. The study found that 41.5% of the test cases containing vulnerabilities could not be identified by any of the tools. Another interesting finding was that there was relatively little overlap in terms of tools' vulnerability detection capabilities; only 7.2% of the vulnerabilities were caught by all five tools and 12.1% were caught by only one (varying) tool. In other words, each tool found mostly different vulnerabilities from the others, so in order to get the best detection coverage one would need to use as many of the tools as possible.

MINESTRONE is an architecture that enables the combination of multiple detection technologies into a single, integrated system, in part via I/O redirection technology. I/O redirection and replay frameworks have been used previously in the context of benchmarking and debugging. One notable effort in this space is Jockey [9], which intercepts calls to non-deterministic system calls and CPU instructions, logs the behavior in a recording phase, and replays it from a log file during the replay phase. However, Jockey does not support deterministic replay of a distributed system. I/O profiler and I/O traces replayer [10] work in user space and use library interposition in order to trace system calls. However, like most record/replay approaches, it records all libc function calls and attempts to replay them exactly as recorded, breaking many programs' execution. Unlike other replay frameworks that do not execute function calls but read effects/values from a recorded log file, R2 [11] focuses on replaying function calls and their side effects. It allows developers to select functions that can be recorded and replayed correctly. However, it requires developers to instrument their code and annotate the chosen functions with keywords to generate replay code stubs.

Our approach, detailed in IV-2, combines two technologies: the first performs fine-grained recording using library interposition and the second streams the recorded data across the network and synchronizes the replay for multiple program replicas. This approach allows specific high-level operations to be interposed and is more flexible (i.e., allowing for different read/buffer sizes during playback). Our approach is also more lightweight, only interposing on those function calls related to I/O with which we are concerned.

III. Hypothesis and Methodology

In this research, our hypothesis is that we can build an architecture combining multiple vulnerability detection and containment technologies to outperform a single, state-of-the-art technology in terms of both true positive and false positive detection rates.

Our methodology involves the construction of the MINESTRONE architecture comprising a variety of vulnerability detection and mitigation technologies, and subsequent experimental testing of the MINESTRONE prototype using a test suite of programs containing vulnerabilities. The MINESTRONE architecture and components will be described in detail in the next section. In the remainder of this section we describe the test infrastructure and corpus.

In order to test the effectiveness of MINESTRONE in detecting and containing vulnerabilities, we have utilized a test infrastructure and test cases developed by an independent test and evaluation team in the context of the IARPA STONESOUP program [6]. The independent test and evaluation team, MITRE, created a test infrastructure that includes a test manager for deploying test cases to multiple hosts, where a test harness runs each test case and collects the results. MITRE also developed a suite of test programs - relatively small, engineered test cases containing memory errors (e.g., buffer overflows/underflows) and null pointer vulnerabilities. For each weakness class, there are multiple test case programs, and for each test case program there are multiple good and bad inputs known as I/O pairs. The bad inputs enable testing of whether the MINESTRONE system and its individual component technologies render the vulnerability in the test case unexploitable (true positive rate). The good inputs enable testing that the MINESTRONE system and components do not alter the original functionality of the test program (false positive rate). A test case, made up of I/O pairs, is considered to be successfully handled by the test harness if all the good I/O pairs result in the correct state (be it files created, return codes, etc.) and all of the bad I/O pairs are reported as containing a vulnerability.

The test suite developed by the independent test and evaluation team contained 226 memory error test cases and 114 null pointer test cases. Each of these test cases is made up of 4 to 10 combined good and bad I/O pairs. A total of 546 distinct I/O pairs were created, each of which may be used in multiple test cases. The first step of our methodology is to define a reference value, the base case. To that end, we run the test cases with good inputs and without any MINESTRONE detection technology, and we observe the output of the test program. Next, we run the same test case with bad inputs and again without any detection technology; we observe the output of the test program and its return status code (e.g., 139 for a program that segmentation faults). This first step in our methodology is a sanity check to establish the ground truth correctness of the test suite with no instrumentation.

The next step in our methodology is to execute each test case and their associated inputs within the MINE-

STRONE system architecture. For each test case, we collect the detection results from each MINESTRONE component as well as the final result returned by the overall MINESTRONE system. We also collect resource utilization information associated with each component technology for comparison purposes. The information collected and the infrastructure that we built for data collection within the MINESTRONE system architecture will be described in detail in the next section.

Finally, the last step in our methodology is to process and verify the results. We will provide these details in Section V on Experimental Results.

IV. Experimental Setup

The high-level MINESTRONE architecture is depicted in Figure 1. The SOUP, i.e.,“unknown” software to be tested, ideally program source code, is initially processed by the composer. (Some MINESTRONE components can operate on Linux ELF binaries.) The program is then dispatched and deployed in multiple virtual execution environments (VEEs) leveraging OS-level virtualization for processing with multiple detection technologies. Each of the VEEs receive the same input, using an I/O redirection component that replicates the same inputs to each respective copy of the program. Diversification occurs in each of the selected VEEs. Finally, the composer analyzes the output of each VEE and provides a vulnerability report to the user. In our experimental setup, the vulnerability report is passed to the test manager for evaluation.

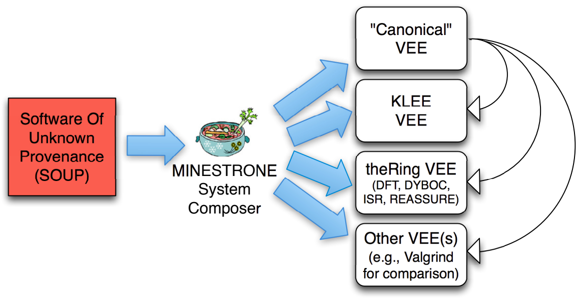
Software Of Unknown Provenance (SOUP)
MINESTRONE System Composer
"Canonical" VEE
KLEE VEE
theRing VEE (DFT, DYBOC, ISR, REASSURE)
Other VEE(s) (e.g., Valgrind for comparison)

Fig. 1. MINESTRONE architecture and workflow

1) Detection Components: In this section we present the core components that comprise MINESTRONE. The detection technology components are key to the vulnerability detection capabilities.

KLEE [12] is a symbolic virtual machine built on top of the LLVM compiler [13] infrastructure.

DYBOC [14] is a source-to-source transformation tool that augments the source code to detect stack and heap-based buffer overflow and underflow attacks.

REASSURE is an error recovery mechanism that allows the software to recover from unforeseen errors or vulnerabilities [15], [16]. It reuses existing code locations that handle certain anticipated errors for unanticipated ones as well.

Data flow tracking (DFT), or taint tracking, is provided via a high-performance data tracking library, libdft [17]. DFT enables the detection of attacks based upon bad inputs, such as those that exploit memory handling errors.

Instruction Set Randomization (ISR) [18] is a detection technology and diversification mechanism that mitigates code injection attacks by diversifying the instruction set used by a program under test at runtime. The attacker’s injected code, relying on gadgets or the native instruction set, will not match the current, transformed instruction set being used and thus fail.

theRing is a customized tool that combines DYBOC, REASSURE, DFT, and ISR into a single execution runtime. It was developed to reduce the overhead required by running separate VEEs for each technology.

2) I/O Redirection: Our framework relies on I/O redirection to provide the same input to all the test programs that are running in isolated containers (VEEs). In order to achieve this we use two complimentary techniques. The first technique utilizes a purpose-built I/O record component to capture all the (pertinent) I/O in a non-diversified “canonical” VEE, and replay it in the VEEs built with MINESTRONE technologies. The record and playback is provided using library interposition to intercept libc function calls that provide input and output operations (e.g., `read`, `send`, etc.).

The library interposer that we have built writes output to a file for later playback. The second technique that we employ allows this output to be streamed live to each of the other VEEs (Klee, theRing, and Valgrind in Figure 1), enabling concurrent execution, using the DUP system [19], a language system for distributed stream processing. After specifying the stream graph, DUP handles the execution of each of the technologies and streams the results back to the MINESTRONE framework to decide which output is “correct”. In Figure 1, the system composer is made up of the test harness and DUP, which controls the execution of the canonical VEE to record I/O, then executes the diversified VEEs, and finally merges the output.

3) Mitigation and Confinement: MINESTRONE builds upon OpenVZ [20] to provide the VEE for each program replica. We also leverage a copy-on-write file system that allows us to discard any unwanted changes due to a vulnerable program. The VEE not only enables confinement but also addresses a specific class of vulnerability known as resource exhaustion [21] by monitoring the VEE behavior from the host. We have the ability to monitor resource utilization to ensure that a program is not using more resources than a given threshold that could be a sign of exploitation, bad programming (e.g., infinite loop) or, in the case of a web server, a Denial-of-Service attack. We developed a tool as part of the MINESTONE framework that executes the test cases in the VEEs, while monitoring and rate limiting three main types of real-time resources in the OpenVZ containers:

CPU usage: we can monitor both user and system CPU consumption as a percentage of total available and as an absolute number of cycles.

Memory usage: we can monitor fine-grained memory consumption of virtual and physical pages allocated.

Network: we can monitor the real-time network traffic that is going to and from the container.

4) Experimental Environment: During the MINESTRONE test and evaluation process, only a single test harness is allowed, meaning only a single "result" can be returned to be evaluated. In our experimental environment we are under no such constraints, so we have chosen to break out each of the individual technologies to determine their standalone detection rates. This allows us to also precisely monitor the time it takes to execute each test case and the comparable resources that are required for each of the technologies. We also collected results on all of the components together to get an overall score. Finally, prior to the test and evaluation, MITRE performed a state-of-the-art analysis using the same test cases. One of the better performing runtime analysis tools that they used for their evaluation was Valgrind [22]; we provide our results using Valgrind as a rough comparison of MINESTRONE technologies to the state of the art.

Our MINESTRONE environment was run on a custom OpenVZ kernel 32-bit CentOS 5 virtual machine (VM) running on a VMware ESX server. The VM was allocated 8GB of memory with 8 CPU cores. For the standalone test results, each technology was allowed to execute in isolation (no other tests running on the VEE).

V. Experimental Results

In performing our evaluation of MINESTRONE using the MITRE test corpus, we ran over 24,000 test cases in 10 different VEEs over approximately one month. In brief, our results show that utilizing MINESTRONE improves detection of memory errors over the state of the art, with a similar resource consumption rate for most components. The one exception is the KLEE component, which uses significantly more resources than the other technologies, with mixed results.

The terms (metrics used by the independent test and evaluation team) in Table I, are defined as:

Processed: A test case is processed when the build with instrumentation was successful in producing an executable binary.

Unaltered Functionality: The test program with instrumentation and good inputs had the expected output.

Rendered Unexploitable: The test program with instrumentation and bad inputs had a vulnerability that has been mitigated by the technology.

Raw Score: The percentage of test cases for which a technology correctly **rendered unexploitable** all bad I/O pairs and had **unaltered functionality** for good I/O pairs, out of all processed and unprocessed test cases.

Processed Score: The percentage of test cases for which a technology correctly **rendered unexploitable** all bad I/O pairs and had **unaltered functionality** for good I/O pairs, out of all processed test cases.

Unaltered Score: The percentage of test cases for which a technology correctly **rendered unexploitable** all bad I/O pairs and had **unaltered functionality** for good I/O pairs, out of all **unaltered functionality** test cases. This is the final score that is used for evaluation.

Table I shows the scoring results from the memory corruption test cases for each of the component MINESTRONE technologies. The highest level result is that our combined MINESTRONE methodology allows us to improve detection over the state of the art by 34.6% (comparing the Combined/Unaltered Score with Valgrind/Unaltered score). Importantly, this result is *not* due to any single mitigation technology outperforming Valgrind, but due to different technologies detecting different memory errors.

In terms of maintaining the functionality of the executable, KLEE and DYBOC are the worst offenders (altering 168 and 98 test cases, respectively). For KLEE, this is due in part to an inability to model memory allocated by libraries outside the main executable. DYBOC fails to properly transform 51 test cases, due to an implementation limit of 65536 total variables, which for undiscovered reasons simply changes the output in the rest of the cases. There are 22 test cases that are altered for all technologies; these test cases relied on a DNS server that was not provided by MITRE.

KLEE provides the highest percentage of correct results, but only for the 53 unaltered functionality test cases. DFT and REASSURE provide the worst detection rates, but also do not alter functionality significantly.

For sake of brevity, we only summarize the scoring results for the null pointer vulnerability test cases. The takeaway from these results is that null pointer errors are relatively easily detected. With the exception of KLEE and DFT, all technologies (including Valgrind) detect a high percentage (greater than 99%) of these errors. Combining all the MINESTRONE technologies provides only a single additional error detected (112 vs. 111), and improves on altered functionality in only two test cases (113 vs. 111). It should be noted that KLEE fails to correctly execute any of these test cases due to library incompatibility. Also, taint tracking is not designed to find null pointer errors, so it's not surprising that DFT detects none of these errors.

Table II compares resource consumption (CPU, peak memory usage and runtime) between the technologies

	KLEE	**DFT**	**ISR**	**REASSURE**	**DYBOC**	**theRing**	**Combined**	**Valgrind**
# Processed	97.69% (221/226)	98.23% (222/226)	98.67% (223/226)	100% (226/226)	100% (226/226)	100% (226/226)	100% (226/226)	100% (226/226)
# Unaltered	23.98% (53/221)	90.0% (200/222)	86.55% (193/223)	90.27% (204/226)	56.64% (128/226)	84.96% (192/226)	90.27% (204/226)	88.50% (200/226)
Raw Score	22.12% (50/226)	4.87% (11/226)	37.61% (85/226)	4.87% (11/226)	45.58% (103/226)	45.58% (103/226)	63.72% (144/226)	47.35% (107/226)
Processed Score	22.62% (50/221)	4.95% (11/222)	38.12% (85/223)	4.87% (11/226)	45.58% (103/226)	45.58% (103/226)	63.72% (144/226)	47.35% (107/226)
Unaltered Score	94.34% (50/53)	5.5% (11/200)	44.04% (85/193)	5.50% (11/200)	80.47% (103/128)	53.65% (103/192)	70.59% (144/204)	53.50% (107/200)

TABLE I. SUMMARY OF SCORING RESULTS ON THE 226 MEMORY CORRUPTION TEST CASES.

	KLEE	**DFT**	**ISR**	**REASSURE**	**DYBOC**	**theRing**	**Valgrind**
Avg. User CPU Jiffies	25086.07 σ 62181.82	2461.68 σ 899.76	3198.04 σ 868.80	1946.90 σ 1164.51	978.61 σ 420.42	4032.69 σ 1420.39	1092.83 σ 1034.03
Avg. System CPU Jiffies	414.55 σ 604.78	364.51 σ 129.08	667.76 σ 190.06	403.65 σ 315.23	201.93 σ 92.66	723.12 σ 244.34	86.46 σ 28.18
Avg. Phys. Mem.	146.99 MiB σ 71.52	36.90 MiB σ 5.32	35.70 MiB σ 5.36	39.87 MiB σ 19.70	30.31 MiB σ 8.21	39.38 MiB σ 5.42	41.65 MiB σ 7.52
Average Runtime	25.59s σ 62.36	4.59s σ 5.47	6.72s σ 13.52	5.49s σ 13.82	3.87s σ 13.52	6.39s σ 7.41	3.7 σ 8.14

TABLE II. SUMMARY OF CPU AND MEMORY USAGE FOR MEMORY CORRUPTION TEST CASES.

for the memory corruption test cases. We computed the average usage across all the test cases that were successfully processed, and present the average and the standard deviation. Most of the technologies use approximately the same amount of resources, with the clear exception of KLEE. KLEE uses between 6 and 10 times the user CPU than any of the other technologies. KLEE also runs longer on average, and consumes more memory. DYBOC is the least resource intensive, using less CPU and memory on average than the other MINESTRONE technologies, and is on par with Valgrind in runtime and CPU. Compared to Valgrind, DYBOC utilizes more than twice the system CPU. This is due to the increase in memory allocations and deallocations that take place as a result of moving stack resources to the heap. Note that the standard deviation given for the runtime is nearly useless; many of the test cases have a built in `sleep` which causes the duration to be inflated, throwing off the average and standard deviation.

Figure 2 compares the physical memory usage over time for a single memory corruption test case. In this case (as is typical, reflected in Table II) we see that all the technologies other than KLEE have roughly the same memory profile, to the point that dintinguishing them is difficult. However, KLEE stands out, peaking at almost four times the memory consumption of the other technologies. It should be noted that the test setup requires around 10 seconds, which is why the test case execution does not begin until after the 11th second.

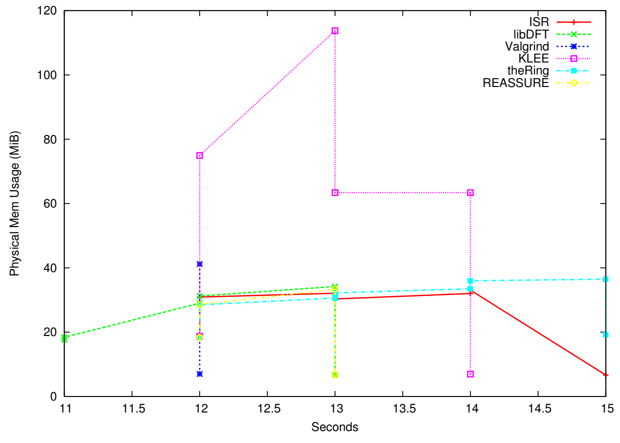

Fig. 2. Graph comparing physical memory usage over time between the technologies. All the technologies other than KLEE use a similar amount of memory; KLEE uses significantly more.

A. Extended KLEE Results

While the results thus far indicate that KLEE is more resource intensive than the other MINESTRONE technologies, the situation is worse than a cursory look at the data indicates. Specifically, we discovered that in many of the test cases that KLEE altered functionality, it was due to a timeout - in other words, the test cases never completed. Each test case in our evaluation was allowed to execute for 200 seconds, after which the test case was killed. In our examination of memory usage data, KLEE increased its memory usage apparently linearly, exceeding 250MB of physical memory, up to 150 seconds of execution (at which point KLEE's built-in timeout begins, but did not complete prior to forced test case termination). For reference, the other technologies lump together in the range of under 50MB physical memory usage.

We looked into memory consumption on a per test case basis in order to see if there were clusters of test cases with similar distinct behavior. We found that most of the test cases (and most of the technologies) again use roughly the same amount of memory (under 50MB), CPU, and time. However, KLEE again stands out in this regard. We looked at a plot with a single point for the memory consumption of each technology with each I/O pair. Again, most of the technologies lump together in a stable pattern, with the exception of KLEE, which occasionally shows a large increase (commonly up to 400MB) in memory usage. Also, KLEE's *minimum* memory usage is about 100MB. We saw similar patterns in CPU usage and runtime (left out for brevity).

B. Extended DYBOC Results

The DYBOC technology relies on multiple components to detect a number of memory defects. The first of these components is a source-to-source transformation based on TXL [23], a source rewriting language and set of tools to perform the actual source translation. DYBOC leverages TXL to transform input source code in two ways. First, any stack allocated variables are changed into heap allocated variables. A simple example of this is replacing the stack variable buf with a heap variable (shown below):

```
char buf[10];
char *buf = malloc(10);
```

The second transformation is to replace memory allocation/deallocation functions (e.g., malloc, calloc, free) with versions of these functions that incorporate memory protections. Specifically, guard pages are placed above and below the actual memory on memory allocation, allowing detection of buffer overflows/underflows.

When running the memory corruption test cases using DYBOC, we noticed a large number of altered functionality test cases (98/226). Further investigation led us to discover a fundamental limitation in the TXL implementation (TXL can handle only 65536 program variables) was causing 51 test cases to not properly be transformed, and thus fail out of hand. Our intuition was that using protected versions of memory allocation/deallocation was more important at mitigating memory corruption bugs than the stack-to-heap transformation. As such, we also ran the test cases using less complex source transformation (only replacing malloc/calloc/free) by leveraging Coccinelle [24]. Coccinelle has the advantage of being a simple-to-understand transformation language.

Using only Coccinelle allowed us to achieve a much higher unaltered functionality rate and caught more total memory corruption errors, but the "unaltered" detection rate also dropped. Thus, we reasoned that there must be many memory corruption test cases that were corrupting stack memory. Since we already determined that there were possible issues with TXL, we turned to another source-to-source transformation tool, CIL [25]. We chose CIL because it comes bundled with a number of example transforms, and a driver (cilly) to apply those transforms in place to C code. The built-in transform we used is called "heapify", and similar to DYBOC is intended to move any stack allocated variables to the heap. After performing the *heapify* operation, we then also applied the memory protection source transform using Coccinelle for the final evaluation.

	DYBOC (Original)	**DYBOC (Coccinelle)**	**DYBOC w/CIL (Coccinelle)**	**Combined**
# Altered	43.36% (98/226)	9.29% (21/226)	31.42% (71/226)	8.41% (19/226)
Processed Score	45.58% (103/226)	47.79% (108/226)	37.61% (85/226)	57.08% (129/226)
Unaltered Score	80.47% (103/128)	52.68% (108/205)	54.84% (85/155)	62.32% (129/207)

TABLE III. SUMMARY OF RESULTS FOR DYBOCS.

Table III details the results of our three different diversified DYBOC containers. One interesting result is that the Coccinelle only version of DYBOC has the highest processed score, and the lowest altered functionality. The data shows that while the CIL + Coccinelle source transformations also achieves a lower altered functionality rate than the TXL version of DYBOC, it correctly identifies fewer memory corruption bugs than the original. The combined results again showcase the utility of MINESTRONE. While one different DYBOC version finds fewer bugs than the original, they also find *different* bugs. As a result, the detection rate for these errors while maintaining functionality increased 25.24% (from 103 to 129).

VI. Lessons Learned

As detailed in the previous section, actually running all of the test cases for this evaluation took less than a month. However, the entire test and evaluation process of the STONESOUP program encompassed more than a year of test suite development by the independent evaluation team and two formal week-long test and evaluation site visits. Suffice it to say, there were a number of lessons that we learned over the course of this experimentation and testing. We provide some of the more valuable takeaways in this section.

1) Symoblic Execution Limitations: Our symbolic execution engine is KLEE [12]. One of the fundamental limitations with this type of analysis is that the state space is broad and the exploration may take too much time, which is why recent research efforts have focused on merging states to enhance speed [26]. In MINESTRONE, to provide a vulnerability assessment in a timely fashion we use KLEE with concrete input. The program is run with its input so that only the path traversed by this input value will be verified. While this is an effective way to attain results based on specific input, it suffers from poor code coverage.

Perhaps the biggest limitation of such a tool is that for it to be applicable to real-world programs, one must provide a model and behavior for all dependencies (e.g., libraries), otherwise the tool fails on any memory allocated outside the tested program. This is a non-trivial task that requires significant effort and expertise, making it difficult to use in enterprise environments where libraries are often selected on a per-project basis. As noted in Section V, KLEE was able to process only a small proportion of even the simplistic test cases provided for the evaluation. This was due to both the slowdown imposed by KLEE and the use of external libraries in the test cases.

As an example of our real-world experience with KLEE, when running GNU grep KLEE imposed a 2700x overhead on the run time, and during the entire execution duration a single CPU core was pegged at 100% usage. The lesson to take away here is that while symbolic execution is a powerful technique for source code (and binary) analysis, it has limitations that keep it from being possible to use in many cases.

2) Diversification: We used several types of diversification techniques, from switching between compiler versions, using different compiler optimization levels and options, altering the OS, and instruction set diversification. The main limitation that we encountered with our software diversification techniques is that the dependencies (provided as part of the base test harness) need to be compatible with the diversified environment. For instance, 32-bit binaries relying on a specific glibc version were provided to us, which precluded us from pursuing more aggressive diversification, such as producing a binary targeting a different architecture (ARM or any non-x86) because of the dependencies. We believe the takeaway is that whenever possible, required dependencies should be built per the end system, and not distributed as part of the test corpus (if so, they should be distributed as source).

3) Test Suite: The test suite we used for our experimentation was provided by MITRE in the IARPA STONESOUP program [6]. This test suite was developed by engineers by hand and contained unintended vulnerabilities that caused "false" positives (which were actually true) or undesired output, causing valid test cases to fail.

For example, in certain test cases a stack variable is declared, and it is assumed to be initialized as all zeroes. We discovered that depending upon the compiler used, and the order in which functions are called, the stack may reuse space for this variable, causing it to be nonzero. If the variable were declared *static* then this code would not cause issues.

Another limitation of the test suite was a lack of documented ground truth. When we received the source code for the test cases and I/O pairs, there was not always a clear indication of *what* the vulnerability was and more importantly *where* it was in the code. This, coupled with the sometimes unintended bugs and vulnerabilities, made our determining whether it was the test case or the technology under test at fault a nearly impossible task. This is why the final scoring was based on test cases with unaltered functionality, as the root cause could not be determined.

The lesson learned here is that for a test suite it would be better to have vetted test case code, modified in specific places to inject vulnerabilities. Building an entire test case from scratch (which may already contain bugs), and then incorporating vulnerabilities by hand makes evaluating the final results that much harder. Also, when releasing a test corpus of code containing errors, the locations of those errors need be released as well, so a determination can be made whether a fault is found in the correct place.

4) I/O Redirection: The subject of using I/O redirection to save and playback program steps is nothing new; as detailed in Section II there are many I/O capture/playback frameworks. We had initially planned on leveraging these existing technologies for MINESTRONE. However, none of those that we were able to obtain (including ioapps and Jockey) actually worked out of the box. Namely, they were difficult to build and once built never actually worked with any of the test cases provided to us. For that reason, we decided to create a simpler, dumbed-down version of our own library interposer for our purposes. Our I/O redirection has some limitations, including non-deterministic execution due to calls to `select`, which are quite common in network servers. The I/O in the test corpus provided to us contains relatively simple test cases, which use mostly single-threaded applications utilizing network, file, shared memory, and X I/O. For these specific cases, our library interposition works well; extending it to more complex test cases will obviously require more effort. The lesson learned is that even a well-researched topic with existing implementations should not be assumed to solve a problem until it is tried.

5) Our Platform: We tried several enterprise products for vulnerability testing; in general they can be helpful to developers but often the sheer number of false positives are overwhelming. Each of these false positives need to be manually analyzed to determine whether the checker is being overzealous or there really is a bug. We have discovered that many of the single-purpose tools leveraged in the MINESTRONE framework are very efficient when focusing on a single, specific type of vulnerability. The lesson that we learned from building and experimenting with our platform is that a combination of tailored tools can provide better detection rates than one single multi-purpose tool. Also, analyzing binaries at runtime in isolated VEEs both protects the host system from compromise and generally does not provide false positives.

VII. Conclusion

In this paper we have presented MINESTRONE: a novel architecture that integrates static analysis, dynamic confinement, and code diversification to identify, mitigate, and contain a broad class of software vulnerabilities. We presented our methodology and experiment focusing on type-unsafe languages. Our results demonstrated that MINESTRONE is successful in combining several technologies in order to improve the detection rate of vulnerabilities while increasing neither the false positive nor the false negative rate. Our results showed an improvement of 34.6% over Valgrind, a state-of-the-art tool, for memory corruption test cases, with similar resource utilization. We plan to continue extending our framework to handle different classes of vulnerability and experiments with other test suites relying on fault injection framework.

Acknowledgment

This work was supported by the US Air Force through Contract AFRL-FA8650-10-C-7024. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government or the Air Force. The authors would like to thank Angelos Keromytis and the MINESTRONE team. Acknowledgment is also extended to the members of MITRE who worked on the test suite and test infrastructure.

References

[1] (2010) Microsoft security development lifecycle. [Online]. Available: "http://www.microsoft.com/en-us/download/details.aspx?displaylang=en&id=12285"

[2] (2013) Cert statistics. [Online]. Available: http://www.cert.org/stats/

[3] (2013) Hp fortity. [Online]. Available: https://www.fortify.com

[4] (2013) Coverity development testing platform. [Online]. Available: http://www.coverity.com/

[5] A. D. Keromytis, S. J. Stolfo, J. Yang, A. Stavrou, A. Ghosh, D. Engler, M. Dacier, M. Elder, and D. Kienzle, "The minestrone architecture combining static and dynamic analysis techniques for software security," in *Proceedings of the 2011 First SysSec Workshop*, ser. SYSSEC '11, 2011, pp. 53–56.

[6] (2011) Securely taking on new executable software of uncertain provenance (stonesoup) program. [Online]. Available: http://www.iarpa.gov/Programs/sso/STONESOUP/stonesoup.html

[7] J. Merced. (2012) Source code analysis tool evaluation. [Online]. Available: http://www.iarpa.gov/stonesoup_Merced_DHSAWGbrief.pdf

[8] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large intel®itanium®programs with dynamic instrumentation," in *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, 2004, pp. 81–92.

[9] Y. Saito, "Jockey: A user-space library for record-replay debugging," in *In AADEBUG05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*. ACM Press, 2005, pp. 69–76.

[10] (2012) Io profiler and io traces replayer. [Online]. Available: http://code.google.com/p/ioapps/

[11] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang, "R2: an application-level kernel for record and replay," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08, 2008, pp. 193–208.

[12] C. Cadar, D. Dunbar, and D. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08, 2008, pp. 209–224.

[13] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.

[14] S. Sidiroglou, G. Giovanidis, and A. D. Keromytis, "A dynamic mechanism for recovering from buffer overflow attacks," in *Proceedings of the 8th Information Security Conference (ISC)*, 2005, pp. 1–15.

[15] G. Portokalidis and A. D. Keromytis, "Reassure: A self-contained mechanism for healing software using rescue points," in *In: Proceedings of the 6th International Workshop on Security (IWSEC)*, 2011, pp. 16–32.

[16] S. Sidiroglou, O. Laadan, A. D. Keromytis, and J. Nieh, "Using rescue points to navigate software recovery (short paper)," in *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, 2007, pp. 273–278.

[17] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: practical dynamic data flow tracking for commodity systems," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, ser. VEE '12, 2012, pp. 121–132.

[18] G. Portokalidis and A. D. Keromytis, "Fast and practical instruction-set randomization for commodity systems," in *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, 2010, pp. 41–48.

[19] K. C. Bader, T. Eißler, N. Evans, C. GauthierDickey, C. Grothoff, K. Grothoff, H. Meier, C. Ritzdorf, and M. J. Rutherford, "Dup: A distributed stream processing language," in *Proceedings of the IFIP International Conference on Network and Parallel Computing (NPC 2010)*, ser. LNCS 6289, 2010, pp. 232–246.

[20] (2012) Openvz linux containers. [Online]. Available: http://www.openvz.org/Main_Page

[21] (2008) Cwe-400: Uncontrolled resource consumption ('resource exhaustion'). [Online]. Available: http://cwe.mitre.org/data/definitions/400.html

[22] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '07, 2007, pp. 89–100.

[23] J. R. Cordy, "The txl source transformation language," *Sci. Comput. Program.*, vol. 61, no. 3, pp. 190–210, Aug. 2006.

[24] J. Brunel, D. Doligez, R. R. Hansen, J. L. Lawall, and G. Muller, "A foundation for flow-based program matching using temporal logic and model checking," in *The 36th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, Savannah, GA, USA, Jan. 2009, pp. 114–126.

[25] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "Cil: Intermediate language and tools for analysis and transformation of c programs," in *Proceedings of the 11th International Conference on Compiler Construction*, ser. CC '02, 2002, pp. 213–228.

[26] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, ser. PLDI '12, 2012, pp. 193–204.