Piston: Uncooperative Remote Runtime Patching

Christopher Salls
University of California, Santa Barbara
salls@cs.ucsb.edu

Yan Shoshitaishvili
Arizona State University
yans@asu.edu

Nick Stephens
University of California, Santa Barbara
stephens@cs.ucsb.edu

Christopher Kruegel
University of California, Santa Barbara
chris@cs.ucsb.edu

Giovanni Vigna
University of California, Santa Barbara
vigna@cs.ucsb.edu

ABSTRACT

While software is now being developed with more sophisticated tools, its complexity has increased considerably, and, as a consequence new vulnerabilities are discovered every day. To address the constant flow of vulnerabilities being identified, patches are frequently being pushed to consumers. Patches, however, often involve having to shutdown services in order to be applied, which can result in expensive downtime. To solve this problem, various *hot-patching* systems have been devised to patch systems without the need for restarting. These systems often require either the cooperation of the system or the process they are patching. This still leaves out a considerable amount of systems, most notably embedded devices, which remain unable to be hot-patched.

We present Piston, a generic system for the remote hot-patching of uninterruptible software that operates without the system's cooperation. Piston achieves this by using an exploit to take control of the remote process and modify its code on-the-fly. Piston works directly on binary code and is capable of automatically counter-acting the destructive effects on memory that might be the result of the exploitation.

ACM Reference format:
Christopher Salls, Yan Shoshitaishvili, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. 2017. Piston: Uncooperative Remote Runtime Patching. In *Proceedings of ACSAC 2017, Orlando, FL, USA, December 4–8, 2017,* 13 pages.
https://doi.org/10.1145/3134600.3134611

1 INTRODUCTION

The modern world is run by interconnected software. Software handles our communications, manages our finances, and stores our personal information. In addition, with the rise of the Internet of Things (IoT), the number of embedded devices running complex software has skyrocketed [49]. In fact, the number of bugs found in software has been increasing over time [43]. Leveraging these bugs lets an attacker perform actions ranging from the theft of money or data to, in the case of the Internet of Things, influence the physical world.

The common approach to remedying buggy software is *patching*. However, patches suffer from very slow *adoption* by users, in part because many patches require system restarts to be applied or to take effect [30, 32]. In the IoT world, the situation is even more problematic, as device vendors often fail to incorporate effective and easy-to-use means to update their products. As a result, even when a vulnerability is found and publicly disclosed, it is difficult (or even impossible) for users to install these patches. Finally, the IoT market is a volatile space, with vendors entering and leaving the ecosystem. This means that a vendor might not be around anymore while its vulnerable devices are still connected to the network.

In the best case, when a vulnerability is discovered, the responsible software vendor will simply develop a patch and push it to its users to secure their devices. Unfortunately, this scenario does not always play out. As mentioned above, a device might lack update functionality, users might not understand how to apply patches (for example, when firmware must be flashed), or the software vendor is no longer present. In all these cases, we would like a mechanism that is able to "force" a patch onto the vulnerable system and fix the vulnerability.

In this paper, we present a technique, called *Piston*, that leverages the presence of bugs to automatically *patch* a system as the result of exploiting these vulnerabilities. By leveraging an exploit to patch software, Piston has the unique ability to patch applications without direct privileged access or, in fact, *without any access to the host at all*. Of course, exploiting a vulnerability in a target process and using this access to patch the underlying vulnerability raises a number of questions and poses significant challenges:

First, not all bugs can be used for patching – it must be possible to take control of the victim process. "Fortunately," a significant portion of bugs manifest as *memory corruption* leading to *control-flow hijacking* [4]. Our intuition is that, aside from taking control of a process for nefarious purposes, a control-flow hijack can be leveraged to achieve remote hot-patching of buggy software.

Second, leveraging an exploit to forcefully take control over a process can have adverse effects on the execution of this process, such as causing a crash. In some cases, this would not be a problem. That is, Piston could take control over the process, update the vulnerable application on the system (on persistent storage), and restart it. Unfortunately, this approach does not always work. One problem is that the running process might not have the privileges to write to the permanent storage, and hence, cannot make the patch persistent. Another problem is that the software might control a critical process, and interrupting its execution has unintended and unwanted consequences. Hence, it is critical that we perform the patch in a way that allows the process to continue its execution without interruption (longer than it takes to patch the running code) or even a crash.

Previous work has introduced the idea of *hot-patching*; a system able to apply patches to software while the software is running. Such systems have been developed for vehicles [27], kernels [37], user-space software [38], and, in general *uninterruptible systems*, or systems where correctness depends on their continuous execution. However, these approaches typically have two requirements: fore-planning on the part of the author, and privileged access to the computer running the software. For example, kernel hot-patching systems, such as KSplice [37], require a custom kernel module to be loaded, which requires administrative privileges. Unfortunately, a large amount of software does not meet these requirements. User-space software rarely supports updates without a restart, and many embedded devices do not give the user necessary permissions. To remedy such situations, a new approach to patching is required. Specifically, Piston uses novel applications of binary analysis to identify and automatically repair data that is corrupted as part of the exploit. That is, our system exploits a vulnerability to take control of the running process, repairs the damage that this exploit has caused, then patches the bug in the code, and finally lets the (now secure) process continue to execute.

Here we will talk about four distinct applications of Piston.

Patching uncooperative systems. Certain systems, such as embedded devices, require that updates be created and distributed by the device manufacturer. This poses a problem to end users: these patches are often only provided for a limited period of time, are produced very slowly, or are never produced at all (in fact, many smaller embedded devices lack any sort of update mechanism). Even when patches are distributed, it might be inconvenient to apply them. Some devices need to be physically connected to a computer to apply the update, and reboots are standard in almost all cases. Piston allows these devices to be updated remotely, as long as the original firmware has a vulnerability that can lead to code execution. The systems do not have to be designed to be hot-patched with Piston, unlike with prior approaches for hot-patching embedded systems.

Patching continuity-critical systems. In some applications, downtime can be prohibitively costly, or even mean the difference between life and death. A couple examples of these applications are critical infrastructure components and medical devices. If a vulnerability is discovered in such systems, it may take significant time before an update can be applied in a safe, scheduled maintenance window. As such, systems that have not been developed with hot-patching in mind, may remain vulnerable to exploits for quite some time as the maintainer prepares for the downtime to apply the patch.

Piston can instead use this vulnerability (if it leads to code execution) to provide the update while the system is running. This can reduce the potentially dangerous delay, as well as preventing the need to schedule emergency maintenance downtime in such cases.

Emergency patching. As more of our personal and business dealing moves online, security becomes paramount. Whereas compromises may have been simply embarrassing to an organization a decade ago, today they can cause serious damage to companies. Thus, organizations must patch software flaws as soon as possible. However, many organizations struggle to roll out security updates. If properly used, Piston could make them easier. Piston could be used as a first-stage emergency patching system. In our example detailing a patch of `NGINX` in Section 6, a company running internet-facing `NGINX` services could scan their entire network and use Piston to apply ephemeral (in-memory) emergency patches to every vulnerable host to tide them over until a permanent patch can be deployed.

Helpful worms. Users and businesses are slow to update devices, often leaving machines vulnerable long after patches are available. For example, the Wannacry ransomware exploited a flaw for which a patch was available three months earlier [22]. Previous work has explored using "helpful" worms to apply patches on a large scale, without users or admins needing to apply the patch [11]. One example is the *Welchia worm*, known for removing the harmful Blaster worm and patching the device. Piston could enable the creation of these helpful worms, even when the vulnerable process does not have enough privileges to apply the official patch, by applying the patch in-memory.

In this paper, we describe Piston's approach and detail its implementation atop an existing open-source binary analysis framework [44]. We discuss situations in which Piston can operate automatically and semi-automatically, and evaluate its efficacy on a handful of binaries from DARPA's Cyber Grand Challenge with documented vulnerabilities. We target stack overflows in these binaries and show Piston's effectiveness at automatically remotely patching through a memory corruption exploit. Additionally, we demonstrate Piston's applicability by remotely patching NGINX 1.4.0 against CVE-2013-2028 [1], using that same vulnerability to achieve remote code execution. We do this to show that Piston can be used on complex, real-world binaries with very little analyst intervention.

In summary, this paper makes the following contributions.

Remote hot-patching. We detail our design for an automatic, remote hot-patching system, called Piston, which generates patches from compiled binaries.

Recovery from an exploit. We introduce novel techniques to automatically recover a program's state and continue execution after an exploit.

Evaluation. We use a set of binaries from the DARPA Cyber Grand Challenge to evaluate Piston's effectiveness at achieving automated remote patching through the exploitation in addition to evaluating Piston's application to real-world, commonly deployed software, such as NGINX.

2 OVERVIEW

Piston is not the first approach to patching computer software at runtime, a process known as *hot-patching*. In this section, we will give a general overview of Piston, and its novelties, before moving on to describe the individual steps in detail in the next section.

Unlike previous work, Piston is designed to patch uncooperative systems *remotely*. As the systems it targets are not designed to be patched in this way (hence *uncooperative*), this patching requires a level of remote access unintended by the authors of the software being patched. Piston achieves this access through the use of an *exploit*. This adds two significant challenges to the patching process. First, unlike existing hot-patching systems, patching must be performed during the exploitation of the vulnerable process, rather than selecting easy patch points. Second, the exploitation of the target process frequently damages that process' memory space. To allow the program to continue executing, Piston must *repair* the memory space of the program after the patch is applied, all while the software is running.

Unlike some prior work, Piston functions directly on binaries, with no access to source code. This allows Piston to work on proprietary software without source code from the vendor, but also makes its work more complicated, as a substantial amount of relevant information is lost when a binary is compiled.

Piston is rather complex, and we introduce a number of terms throughout this paper to simplify explanation. To aid the busy reader, we have also compiled a glossary, containing definitions of these terms, in Appendix A.2.

Piston has four pre-requisites for its operation:

Original binary. This is the binary program that is currently running as the remote process or system.

Replacement binary. This is the "patched" binary. The remote process will be functionally updated to this version of the binary after Piston's operation.

Exploit specification. Piston expects a description of how to trigger a vulnerability in the remote process. This specification must be able to achieve code execution in the remote process, which Piston will use to apply the patch. The exploit is expected to bypass common mitigations such as ASLR and NX if they are used on the target system.

Remote configuration. To properly model the environment of the remote process, Piston needs to have a specification of its configuration. For example, if the remote process is an `nginx` web server, its configuration file must be provided.

Given these inputs, the approach has three major steps:

1. Patch generation. Given its inputs, Piston performs in-depth static analysis of the binary to identify the "patch" that needs to be applied in the memory of the remote process. This is done by leveraging binary diffing techniques, which is discussed in detail, in Section 3.

2. Repair planning. Unlike traditional hot-patching systems, Piston *exploits* a process in order to patch it. Thus, Piston faces a unique challenge: in the course of exploiting the remote process, the memory state of the remote process might be damaged.

Piston has the capability to automatically generate a routine which repairs the corrupted state of a process if it was exploited with a stack-based buffer overflow. For cases which Piston cannot repair automatically, including other types of exploits, Piston will require the analyst to provide a *repair routine* that should repair the parts of the process' memory that Piston is unable to restore. Piston, can report the parts of the state that were corrupted to the analyst to aid in the creation of the repair routine. In our evaluation (see section 6) we show that this repair routine can be automatically generated in the majority of stack-based buffer overflows which we tested.

Piston may also require a *rollback* routine that undoes the partial effects of functions that were interrupted by the exploit. In the case where a patch involves making a change to a structure definition, Piston requires an analyst to supply a *state transition routine*. This routine should be responsible for updating all instances of the structure in the target's memory to abide by the newly patched-in definition.

We talk in-depth about cases where Piston can fully automatically repair the state and cases where analyst intervention is necessary in Section 4.

3. Remote patching. Piston uses the exploit specification to craft an exploit to inject the *patcher core*. The patcher core, running in the remote process, retrieves the patch information, a state transition routine, a rollback routine, and a repair routine. Piston may deem any one of these routines to be unnecessary to the hot-patching process, with the exception of the state transition routine where an analyst is responsible for judging its necessity.

Piston uses the patcher core to then apply these received routines in turn. After this is completed, the execution returns to the now-patched remote process and Piston's operation is complete. In-depth details of the patching step are in Section 5. When it terminates, the remote process will be running a codebase that is functionally equivalent to the patched binary. A step-by-step example is provided in Appendix A.1.

3 PATCH GENERATION

Piston receives, as input, the original binary representing the target process and the replacement binary to which the target process should be updated. Given these binaries, it must identify specific patches that must be applied in order to accomplish this update.

Similar to other systems, such as Ksplice, Piston applies patches on a function level rather than replacing the entire binary in the remote process. If a function is updated in the replacement binary, its counterpart in the remote process (running the original binary) will be replaced at runtime. If a function is found to be *unique* to the replacement binary, it will be added to the remote process.

Additionally, in the updated binary, addresses of code and data will usually change. Therefore any references to code and globals must be updated in the replacement functions. Piston will fix the references to point to those in the currently running process.

Piston's preprocessing works in several stages:

(1) Piston matches updated functions between the original and replacement binaries. The matches are filtered to eliminate superficial differences.
(2) Piston chooses a location in the memory space of the remote process, in which, to place remaining updated functions. These functions are "fixed up" to allow them to function in the memory space of the remote process and run in the context of the original binary.

The output is a *patch set* (represented as a diff of memory) that Piston will apply to the remote process in the remote patching step.

3.1 Function Matching

First, Piston must identify the functions that need to be updated or added to the remote process. This requires Piston to understand which functions in the original binary correspond to which functions in the replacement binary. We leverage existing binary diffing techniques [17] for this step, allowing us to support the correlation of functions even when there are no symbols in the binaries. These techniques work on the control flow graph, so they are robust to small compiler artifacts.

At the end of this stage, Piston generates a set of pairs of matching functions and a set of introduced functions. Additionally, a candidate set of original function to replacement pairs is constructed by checking for differences in the content of matched functions.

Piston's initial candidate set of updated functions contains some false positives. This is because any change in the length of code will cause addresses to be different in the replacement binary; in turn, the differing addresses will show up as changes in the operand of instructions. We consider all references to the same code or data between a pair of matched functions to be superficial. An example of a superficial difference is shown in Figure 1.

Thus, Piston *filters* this set of updated functions to remove superficial changes. If an updated function contains only superficial changes,

```
push    ebp
mov     ebp, esp
sub     esp,0x18
mov     eax, 0x804a02c
:
0x804a02c  "Hello %s"
```

```
push    ebp
mov     ebp, esp
sub     esp,0x18
mov     eax, 0x805a084
:
0x805a084  "Hello %s"
```

Figure 1: Superficial difference example

we discard it and its match from the candidate set. The remaining members of the candidate set, along with the introduced functions, are the ones that Piston will patch into the new binary.

3.2 Replacement Function Placement

Because Piston replaces individual functions rather than the entire binary, it runs into the challenge of function placement. As previously discussed, replacement functions may be larger than their original counterparts. Because of this, Piston chooses a new address in the executable memory space of the remote process to place replacement functions. This requires an addition function fix-up step: any relative references in the function will need to be updated to compensate for the new location.

New functions or data that were not in the original binary can be resolved by adding the code or data to the new process. Newly added code and data might also have references to other code and data, so it needs to be handled similarly until all references are resolved.

After determining a place for the replacement function in this new area of memory, Piston places a trampoline (direct jump instruction) at the beginning of the old function. Piston checks that the trampoline will fit entirely inside of the first basic block of a function to ensure that execution will never jump to the middle of an instruction. This is useful for several reasons. First, it lets us replace the function while keeping all references to it, such as function pointers and direct calls from other places in the code. Second, if there are any return addresses to code inside the original function on the stack, they remain valid, although the patched code will not be executed until the function returns. Note that this means that Piston can only patch functions that will eventually return, and infinite looping functions, such as a main loop, cannot be patched. This minor limitation is also common among other hot-patching systems.

4 REPAIR PLANNING

Piston achieves the hot-patching of a remote process by leveraging an exploit to achieve code execution in the context of the process, and then using this capability to inject patched code before resuming process execution. Unfortunately, exploits typically cause the *corruption* of the memory space of the remote process, and resuming a process after such corruption can be non-trivial.

For example, during the exploitation of a stack-based buffer overflow, process data on the stack is overwritten with either shellcode or a ROP chain, ultimately leading to the hijacking of control flow by the exploiter. If this memory corruption is not corrected before execution is resumed, the process will simply crash. To remedy this, a *repair* step is required before Piston can resume the patched process.

Piston can fully automatically generate a repair routine in cases of stack-based buffer overflows. This automatic approach uses redundant data in memory and registers to restore the state of the process. In principle, this approach applies to any corruption, not just that on the stack. However, empirically, we have not found an adequate level of data redundancy in other classes of exploits, and therefore, require the analyst to provide the repair routine if the exploit is not a stack-based buffer overflow. We discuss the redundancies inherent in stack data in Section 4.3 and the limitations in repairing other corruption in Section 7. We focus on buffer overflows in Piston's current implementation, as they still represent the third most common type of vulnerability in *all* software [4]. Furthermore, a recent analysis of trends in CVE's found that buffer overflows rank the highest for severity, and that buffer overflows are the second most common vulnerability that applies to binary software, behind denial of service vulnerabilities [13].

Piston carries out an offline analysis of the original binary and the exploit specification to assess the damage that an exploit causes and creates such a repair plan. This analysis is done *off-line*, before the patching process itself, and the repair plan is applied to the remote process after it is patched. The on-line patch application step is discussed in Section 5. Piston assumes that the exploit it will use to patch the remote process will hijack execution either partway through a function or at the return point of a function. We term this the *hijacked function*, and reason about exploitation after-effects as they relate to this function. We name the function that calls the hijacked function the *caller function*.

Piston can restart the remote process after patching if the following conditions are met:

(1) The hijacked function either completes successfully (i.e., the exploit does not influence its operation and simply hijacks the control flow when it returns), or its effects (such as memory writes) can be analyzed and *undone* and the function can be restarted. In the former case, Piston can simply return to the caller function after the remote process is patched. However, in the latter case, the effects of the hijacked function, such as the modification of memory and registers, must be undone. After undoing this modification, Piston can return to the *call-site* of the hijacked function, and trigger its re-execution after the patching is complete.

(2) Any state of the caller function that was corrupted (such as local variables within in the stack frame) can either be recovered or is not needed after the patched process resumes. If the caller function has corrupted state that cannot be recovered, Piston can try to treat the caller function as *hijacked* and, instead, undo its effects and try to restart it. In this case, Piston's recovery process is repeated with the prior caller function being the new hijacked function and the *caller* of the original caller function to be the new caller function.

To meet these requirements, Piston creates a repair plan that includes two routines that will be executed inside the remote process after it is exploited. These routines are the *rollback routine*, which will undo the actions of the hijacked function (if necessary), and the *repair routine*, which will restore the local state of the caller function to be non-corrupted.

Automatically generating these routines represents a significant challenge, and there are two cases when manual analyst intervention might be required. First, depending on the complexity of the hijacked function, Piston might be unable to automatically undo its effects. In this case, the analyst must manually provide the *rollback routine* that will be run in the remote process before the hijacked function is restarted.

Second, the exploit might cause irreparable damage to the caller function's state. In this case, Piston provides two options to the analyst: the analyst can manually provide a *repair routine*, or Piston can attempt to undo and restart the caller function as well. To do so, it moves further up the callstack, classifying the caller function as the new hijacked function and that function's caller as the new caller function and repeating its analysis.

Piston creates the repair plan in three steps:

Exploit effect reconstruction. To reason about the state of the remote process after exploitation, Piston carries out the exploit against the original binary in an instrumented environment. The trace that is created during this step is used in further analyses.

Hijacked function analysis. Piston analyzes the exploit trace to determine whether the hijacked function had successfully completed its work. If the hijacked function was interrupted, Piston must annul the function's effects and restart it after the patch completes. To understand how to properly undo the effects of the function, Piston performs an in-depth analysis of the function using symbolic execution techniques.

Caller state recovery. Next, Piston determines the extent of state clobbering outside of the hijacked function's stack frame by analyzing the exploit trace. It attempts to create a *state repair plan* for this damage, leveraging symbolic execution of the caller function to identify uncorrupted parts of the state that can be used to restore corrupted values.

Different types of exploits cause different damage to the remote process. For example, a simple pointer overwrite might not require much memory repairing, whereas a stack overflow can corrupt much of the stack. In its current state, Piston can automatically create a repair plan for memory corruption resulting from most stack-based buffer overflows. Piston automatically detects corruption resulting from stack-based overflows as well as heap-based overflows, but automatically supporting corruption detection for additional exploits simply requires a routine to recognize the corruption they cause (i.e., expanding the processes described in Sections 4.1.1 and 4.1.2).

4.1 Exploit Effect Reconstruction

Piston generates an *exploitation trace* to reason about the damage that the exploit will cause to the remote process. The exploitation trace is created by executing the original binary (configured with the remote configuration), using the exploit specification as input. During the trace, control flow transitions and writes to and reads from registers and memory are recorded for future analysis.

4.1.1 Detecting the Exploitation Point. To understand what repairs are needed after exploitation, Piston must classify memory writes based on whether or not they are a result of the exploit or of the intended operation of the binary. Piston does this by identifying the *exploitation point*. Intuitively, the exploitation point is a point in the trace after which the process can no longer be considered to be operating properly.

For stack-based buffer overflows, Piston uses a simple heuristic to identify this point: Piston tracks all saved return addresses and callee-saved registers throughout execution. When one of these is overwritten, Piston assumes that it has identified the exploitation point. The function where this exploitation point takes place, is the *hijacked function*.

For heap-based overflows, Piston tracks calls to heap allocation and deallocation functions such as `malloc()`, `realloc()` and `free()`. [1] During the *exploitation trace*, Piston keeps a list of the heap buffers and updates it at every call to these functions. At every write to the heap, Piston checks whether or not the address resides in one of these buffers. If the address does not reside in any such buffer, it is assumed that the exploitation point has been identified.

One caveat of these heuristics is that Piston cannot identify the exact exploitation point for exploits which perform the overflow entirely within a stack frame, or within a struct on the heap. Although advanced type analysis can automatically infer the data types and structure of objects and stack frames [26] [36], such analysis is out of the scope of this paper. Piston can be extended with additional routines in order to support automatic detection of corruption of other exploits such as these.

4.1.2 Identifying Corruption. Once the exploitation point has been identified, Piston can determine the parts of the program state that were corrupted. Again, a heuristic specific to buffer overflows is leveraged: Piston marks as "corrupted" all data that was written to the buffer that was overflowed. This step is done *retroactively* by analyzing all of the writes to memory that occurred.

Piston uses a simple heuristic to identify buffers: it assumes that all writes that are initiated by the same instruction (not the same invocation of that instruction, but all invocations) are writes to the same buffer. This approach is inspired by the buffer detection proposed by MovieStealer [50], which groups buffers by *loops* instead of instructions. Piston marks all writes by the same instruction during the invocation of the hijacked function as writes to the same corrupted buffer. We term this instruction the *overflow instruction*.

4.1.3 Exploitation Trace Soundness. It is possible that the analysis results might not perfectly match the state of the remote process during exploitation. For this reason, Piston augments the trace with more general analyses in other steps and only assumes that two pieces of information from the trace are accurate:

(1) The exploit will overflow the buffer by the same number of bytes in the exploitation trace as it will when run against the remote process.
(2) The hijacked function and its caller function will be the same on the remote process as in the exploitation trace.

We have not seen a case that violates either of these assumptions, but it is a theoretical possibility.

4.2 Hijacked Function Analysis

Having identified the hijacked function and the range of the corrupted data, Piston must next determine whether the hijacked function terminated successfully or whether it needs to be restarted.

Conceptually, the determination of whether the hijacked function terminated successfully is simple: Piston considers the function as having successfully completed if it can show that no action was taken based on corrupted data. This happens fairly frequently: modern compilers tend to avoid placing local variables after buffers in memory, since doing so would allow the local variables (instead of just the return address) to be overwritten by a buffer overflow, potentially allowing the attacker to influence program behavior even before the

[1] In statically linked binaries, such as firmware, an extra step is necessary identify these functions as they may not contain symbols. We use test cases, comprising of input states and expected outputs to identify these functions as described in [44].

function returns. However, in cases where this is not the case (either because the compiler did not choose such a placement or because there is more than one buffer on the stack), we consider the hijacked function's operation to have been *interrupted*, and Piston must undo the corrupted effects and restart the function after patching.

4.2.1 Checking for Successful Completion.

Functions have memory that can be written to without influencing the operation of the remainder of the program; we call this memory *scratch space*. Scratch space is considered to be the local stack frame as well as any memory regions which are freed before the return site of the function. Data in these ranges will not be used outside the function in a well-formed program. Other data, such as globals, heap data which is not freed inside of the function, and return values may influence the remainder of the program and are not considered scratch space.

Piston determines the successful completion of the hijacked function during the dynamic analysis of the offline exploitation trace. At the exploitation point, the corrupted data range is marked as tainted, and the taint is tracked through the remainder of the function. If any branch is influenced by tainted data or if tainted data is written outside of the scratch space, then the function is considered to have not completed successfully. We filter out the restoration of callee-saved registers at the end of the function, as these are considered part of the state of the caller and will be restored later.

4.2.2 Checking for Repeatability.

When Piston is unable to prove that the hijacked function completed successfully, it will check if its execution can simply be repeated after the remote process is patched. The hijacked function can be safely restarted if all of the inputs (i.e., values the function reads from memory or registers) to the interrupted invocation can be recovered. If these inputs can be recovered, the hijacked function can be re-executed in the same context as its interrupted invocation and will carry out the same actions.

Piston groups the inputs that a function receives into three categories: *local state data*, which is passed to the hijacked function on the stack or in registers, *global state data*, which is retrieved by the hijacked function from the heap or global memory, and *environment inputs*, which are retrieved through system calls. The hijacked function is considered *repeatable* if these conditions hold:

Local state data is recoverable. Arguments on the stack, which might be clobbered during the overwrite itself or by actions taken by the hijacked function after exploitation, must be recoverable, as must arguments passed to the function through registers. This recovery is explained in Section 4.3.

Global state data is recoverable. All data in registers and memory that the hijacked function reads must be recoverable. This means that the hijacked function cannot irrecoverably overwrite its inputs.

System calls are repeatable. The system calls invoked out by the hijacked function must be repeatable. System calls that cannot simply be re-executed, such as `unlink` (since, after the first call, the file will no longer exist), violate this condition.

The first condition will be checked during the caller state recovery step. To check the latter two conditions, Piston collects a list of all memory accesses and system calls in the exploitation trace, which can then be checked for any violations to the repeatability conditions.

To detect changes to the global state, the list is analyzed to build a set of any potentially corrupted global state by the original run of the hijacked function. After this, the list is analyzed again to see if any of the corrupted global state can be used as an input to the repeated invocation of the hijacked function. Conceptually, this happens when the hijacked function reads in some global value before writing to it (for example, incrementing a global counter). The second invocation will use the value corrupted by the first, resulting in an inconsistency in execution between the interrupted invocation and repeated invocation of the hijacked function.

Piston will attempt to undo simple global changes where no dereference of data takes place. We use under-constrained symbolic execution (UCSE), an extension of dynamic symbolic execution that enables the analysis of functions without the requirement of *context* [41]. UCSE works by identifying memory dereferences of pointers that are unknown due to missing context (for example, a pointer that would have been passed as an argument) and performs on-demand memory initialization to allow the analysis to continue.

Piston explores the hijacked function with UCSE, *ignoring* the context from the exploitation trace to avoid under-approximating the remote state. If UCSE can determine how a global value will change during execution of the hijacked function, then Piston can recover the value automatically. Note that, like other techniques based on symbolic execution, UCSE can succumb to path explosion. When this occurs, Piston will be unable to automatically recover changes to global state. If any changes to global state are detected that Piston is unable to recover, analyst will be required to provide a *rollback routine* to undo the effects of the interrupted execution.

After checking for changes to the global state, Piston carries out an analysis of system calls. Specifically, it checks for system calls that might not be repeatable. For example, if the interrupted and repeated invocation of the hijacked function both try to `unlink` the same file, an inconsistency between their executions will arise. Because Piston does not have a complete model of all system calls, it presents these lists to the analyst for review. If any system calls are deemed not repeatable, then the analyst must provide a rollback routine to undo the effects of the system calls.

4.3 Caller State Recovery

Regardless of whether the hijacked function has successfully run or needs to be restarted, the state of the caller function must be recovered. Although the general problem of restoring registers and memory to the state of the execution before the overflow is undecidable, we have found that there is often enough data remaining to recover the original state. Our key insight is that, due to the way programmers write source code and compilers compile it, the stack frame and registers of a function often contain *redundant data*, which can be used to restore the corrupted data. In our case, this means that the value of a corrupted stack variable or register can often be determined as some equation of other stack or register values.

```
mov      eax, [ebp+var_14]
mov      edx, [ebp+var_8]
sub      eax, edx
mov      [ebp+var_3C], eax
call     hijacked_func()
```

Listing 1: An example showing where stack variables have redundant information.

Before delving into Piston's approach to state recovery, we provide and briefly discuss an example of such redundancy in Listing 1. Assume the programs instruction pointer is currently at line 5. The stack variable `var_3C` is redundant since it can be computed from the other stack variables, specifically, `var_3C = var_14 - var_8`. Thus, if the overflow clobbers `var_3C`, it can be recovered from `var_14` and `var_8`.

4.3.1 Data Filtering. Before recovering corrupted state, Piston must identify *what* state needs to be recovered. If the hijacked function completed successfully, we must restore any stack variables or registers that were corrupted by the exploit and will be used later in the caller function. Additionally, if the hijacked function was interrupted, we must also restore all of the arguments (on the stack and in registers) that are passed to the hijacked function.

As described in Section 4.1, Piston identifies the range of registers and stack variables that were clobbered by the exploit. In fact, not all of these values *must* be recovered. For example, if a callee-saved register is written to immediately after the hijacked function returns, its value after exploitation, whether or not it was corrupted, is irrelevant, and there is no need to restore it. Piston identifies these cases by computing the control flow graph of the hijacked function and identifying accesses to stack variables and registers. Then a dependency analysis is run on the control flow graph to check if any path exists where a corrupted register or stack variable is read before it is overwritten. If no such path exists, Piston marks the register or stack variables as *unused* and filters it from further state recovery steps.

One caveat must be mentioned for stack values. In some cases, the caller function might pass a pointer to the stack as an argument to the hijacked function. Normally, this happens when a buffer or structure resides on the stack and must be used by the hijacked functions. If the hijacked function performs complex operations on this pointer (such as passing it into other functions or system calls), Piston's static analysis is unable to safely recover these effects. Piston makes the assumption that the passed-in pointer points to the *beginning* of the structure and assumes that the hijacked function may have corrupted anything on the stack after this pointer.

At the end of this step, Piston has a *recovery set* of the registers and stack values that must be recovered before the caller function can resume execution.

4.3.2 Data Recovery. Piston recovers state data by analyzing two locations in the caller function: the function prologue and the hijacked function call site.

Generally, functions will initialize several registers in the prologue and use them for the remainder of the function. This is especially true for registers such as the base pointer (i.e., `ebp` on x86), which are typically set at the beginning of a function. The values of registers that are set in this way can often be determined by analyzing the prologue of a function. Likewise, the caller function *prepares* the call-site of the hijacked function by copying its arguments into argument stack variables and registers. Most of the time, these arguments are passed by value and are drawn from other parts of the state, creating data redundancy that can be leveraged to restore their values when they are corrupted by the exploit.

To avoid under-approximations, Piston does not reuse the exploitation trace in the data recovery step. The control-flow path from the trace may differ from the one that will be executed on the remote server. For example, the remote server may have internal state such as a linked list, which will result in a different control flow than the one in the concrete trace.

To recover data, Piston will analyze two locations with symbolic execution. The first is the start of the caller function up until the first branch. The second location is the callsite of the hijacked function, starting at the earliest basic block from which there is only one path that reaches the call.

Piston analyzes these locations with under-constrained symbolic execution and extracts the relationships between data that must be recovered and the uncorrupted data currently existing in the state. We represent these relationships as equations that produce the recovered values of corrupted data when provided the values of the uncorrupted data. These equations are then examined to verify that all values in the recovery set can be recovered from existing data in the stack.

For example, in Listing 1, when Piston symbolically analyzes the callsite of the hijacked function it will generate a constraint that `var_3C = var_14 - var_8`. If Piston determines that `var_3C` will be overwritten, but not `var_14` or `var_8` then it will determine that `var_3C` is recoverable.

If all values in the recovery set can be recovered from existing data on the stack, Piston saves this set of equations as the *repair routine*. Otherwise, Piston requires the analyst to provide a partial repair routine that recovers corrupted values that are still missing. The repair routine will be executed after the remote process is patched and before it is restarted, as explained in Section 5.

5 REMOTE PATCHING

Until this point, Piston's analysis has been offline: no connection to the remote process has been made. This section describes how Piston uses the provided exploit specification to achieve code execution in the remote process, and applies the results of the offline analyses to repair, patch, and resume the remote process.

The astute reader will recall that, in the previous analyses, Piston recovered the following information for use during the remote patching:

Patch set. In Section 3, we described how Piston identified the set of patches to apply to the remote process to turn it into a functional copy of the replacement binary.

Rollback routine. We introduced in Section 4.2 Piston's strategy for undoing the effects of the hijacked function, if it is determined to have been interrupted by the exploit.

Repair routine. Piston's approach to creating a routine to repair the remote process state after exploitation is detailed in Section 4.3.

While generating this information is complex, the rest of the process is straightforward. Piston executes the following steps, in order:

(1) First, Piston launches the exploit against the remote process. The exploit hijacks the control flow of the remote process and loads a first-stage payload, provided by Piston, which facilitates the execution of the rest of the repair and patching tasks. We call this payload the *patching stub*.
(2) Next, Piston transfers the repair routine to the patching stub. The patching stub executes the repair routine to repair the damage done by the exploit to the remote process state.
(3) If, during the prior offline analysis, Piston determined that the exploit caused an interruption of the hijacked function (i.e., it did not terminate successfully), Piston transfers the rollback routine to its

patching stub and executes it to undo the effects of the hijacked function. As discussed in Section 4.2, in this case, the hijacked function will be restarted after the patching process is complete.

(4) Piston transfers the patch set to the patching stub. The patching stub applies this patch set to the remote process, transforming it into a program that is functionally equivalent to the replacement binary.

(5) Finally, the patching stub returns control to the remote process. If the hijacked function completed successfully, it simply returns to the instruction, inside the caller function, after the call to the hijacked function. Otherwise, control flow returns to the beginning of the hijacked function.

After these steps are completed the remote process has been hot-patched. The remote host is now effectively running the replacement binary, and this has been done without restarting the entire process or performing any permanent changes.

The rest of this section will discuss other minor points relating to Piston's remote patching step.

5.1 Exploit Requirements

Piston has very simple requirements for the provided exploit specification. In short, the specification must describe an exploit that achieves code execution and loads Piston's patching stub. As discussed throughout the paper, if this exploit uses a stack-based buffer overflow to achieve code execution, Piston can often carry out the rest of its work automatically. Otherwise, the user must also provide the rollback and repair routines.

5.2 Optional Patch Testing

Piston supports an optional patch testing step between the offline analyses and the actual remote patching described earlier in this section. If the analyst provides a *test case* to verify that the process has been properly patched, Piston carries out a test run against a locally-executed copy of the original binary. After patching this local process, Piston verifies that the test case passes when run against it. While this is a very straightforward concept, we found that it greatly eased cases when rollback and repair functions had to be provided manually by the user.

5.3 Persistence

Piston is meant to patch the running process *ephemerally* (i.e., without making any actual changes to the filesystem or firmware). While Piston can, during the patching process, execute a user-provided *persistence routine* to persist its changes (for example, by overwriting the original binary on disk), this is not Piston's standard use-case. In fact, we expect that, generally, the process that Piston patches will not have the proper access to write to its original binary on-disk. For example, server processes on Linux almost never have write permissions to their own binaries, and Piston would be running with the same permissions as the server process while patching it.

To patch forking services, Piston would need to apply the patch to the parent process. There are no theoretical limitations which prevent Piston from attaching to, and patching, a parent of the exploited process, granted that our exploited process has permissions to attach to a parent and in addition, that the underlying operating system supports process tracing.

Ephemeral patching itself is a very powerful technique, even without the ability to commit the changes to disk. In Section 6, we showcase how to quickly patch a security flaw in a web server to which the analyst may not have access. That application of Piston does not need to be persistent to be useful. Furthermore, other hot-patching systems such as PatchDroid choose to only patch ephemerally [31].

6 EVALUATION

We evaluate Piston in two ways. First, we test its ability to recover the program state after a stack buffer overflow on all of the applicable binaries from the Cyber Grand Challenge Qualifying Event (CQE). For all CQE binaries with stack buffer overflows, we test if Piston can recover enough state in the caller function, such that the state can be completely restored after an exploit achieves arbitrary code execution. Then, we test Piston's patching functionality on five of those binaries as well as a real-world binary, NGINX 1.4.0 (which is vulnerable to CVE-2013-2028) by creating exploits and using Piston to apply the patch, recover state, and resume execution.

6.1 Dataset

We chose targets for Piston that would allow us evaluate Piston's state recovery methodology. We use binaries from the Cyber Grand Challenge because these represent a large number of binaries containing a wide variety of functionality. Additionally, CGC binaries are guaranteed to have at least one vulnerability as well as a Proof Of Vulnerability (POV) which causes it to crash. As such, these targets are used to test Piston's recovery capabilities in a wide range of binaries. We took the 126 single-binary applications from the CQE and discarded any which did not crash with the provided POV in our testing environment leaving us with 102 binaries. Of those 102, we found that 24 crashed due to an inter-frame stack overflow. We use all 24 for testing Piston's recovery capabilities.

To test the end-to-end patching and recovery from an exploit, we chose five binaries from the above set. For each of these binaries we had to write an exploit which would give us arbitrary code execution. This was required because the provided POVs only lead to crashes, many of which do not crash with control of the instruction pointer.

Along with the CGC binaries, we chose NGINX 1.4.0, which is vulnerable to CVE-2013-2028, to test Piston on a real-world application. NGINX is proves to be an interesting candidate due to its unique architecture among webservers: it initializes a fixed number of worker processes that persist throughout the entirety of the server's uptime. This allows us to patch the individual workers of the NGINX server by repeatedly connecting to the server.

6.2 Recovery Results

To test Piston's recovery capabilities we used the 24 CQE binaries containing an inter-frame stack overflow. We constructed two patching stubs, one that relies on the absence of NX (shellcode stub), and one that bypasses NX using return oriented programming (ROP stub). The shellcode stub is 23 bytes in length whereas the ROP stub is 40 bytes. We trace each of those binaries with their accompanying POVs and use Piston's built-in functionality to identify the exploitation point and the hijacked function in which the overflow occurs. Then we set the overflow amount to that which is needed for each of the patching stubs and check if piston can recover the state.

Piston was able to correctly identify the corruption point in all cases, and was thus able to identify the corrupted data. For the shellcode stub, Piston was able to completely recover the corrupted data for 22 out of 24 binaries. For the ROP stub, which clobbers more bytes of the stack, Piston was able to completely recover the corrupted data for 20 out of 24.

6.3 End-To-End Results

Piston was able to patch all five binaries from our CGC end-to-end dataset as well as patch NGINX, with only two of these six binaries requiring input from the analyst. Only one of these binaries, CROMU_00038, required the analyst to write code. In the other one that required input, NGINX, Piston was unable to generate a rollback function, but the analyst was able to quickly determine that no rollback function was actually necessary.

In the one CGC binary that required the analyst to write code, Piston's patch testing step reported a possible problem. Upon inspection, we discovered that the patched binary sanitizes the input before control reaches the hijacked function, but the runtime patch was restarting the hijacked function with the unsanitized input. By providing a repair function that sanitized the input in memory, the patching was able to proceed as expected.

As part of our experiments we evaluated how much stack space in the caller function could be overwritten before Piston would need to undo and restart the caller function as well. We iteratively increased the amount of overflow until Piston reported that the caller function's frame could not be recovered. These results are shown in Table 1. We found that there was a large variation in the number of bytes in the caller's frame that were recoverable; the results ranged from only four bytes to over three hundred.

6.4 NGINX Patching

In July of 2013, both NGINX version 1.3.9 and 1.4.0 were found to be vulnerable to a stack-based buffer overflow which results from improper handling of HTTP chunked transfer-encoding (this vulnerability was given the label CVE-2013-2028). NGINX is not a simple binary; the source code alone for this version approaches 180,000 lines of code. By successfully patching NGINX through this CVE, we demonstrate Piston's effectiveness and applicability.

We began our evaluation by compiling two versions of NGINX; one version represents the original binary, and the other is the replacement binary. We obtained the original binary by downloading the NGINX 1.4.0 source code and compiling it. For the replacement binary we took the same source code and applied the CVE-2013-2028 patch file provided by nginx.com [3]. Next we developed an exploit specification targeting the vulnerability. Our exploit specification is simply an exploit script which gets to shellcode execution on an NGINX worker process; many exploits for this particular CVE can be found online [2, 29].

While Piston was analyzing the hijacked function, it determined that the function was interrupted and would need to be repeated. Upon determining that the hijacked function must be repeated, Piston identified small changes which would be made to the global state of the process on a repeated call of the function. Piston was unable to generate a rollback routine for these particular changes, so deferred the creation of a rollback routine to the analyst, highlighting the changes made during the repeat. In a matter of seconds, we, as analysts, can see that the effects of a repeat call are inconsequential, and inform Piston to carry on without rollback.

Next, Piston was able to successfully determine that four bytes of the caller's state were destroyed, as a result Piston then generated a repair routine which recovered these four bytes. However, for the sake of evaluation, we show that 28 bytes of the caller's state could have been corrupted without hindering Piston's ability to generate a repair routine automatically.

After these steps, the brunt of the analysis is complete. Piston now executes the patcher using the exploit specification provided to first get shellcode execution. With shellcode execution Piston then reads in and executes the repair routine generated earlier. Then, Piston's shellcode performs the patching process and soon reports that the patching is complete.

We verify that the NGINX web server is still running by manually making a request with a browser. Next, we verify that the server has successfully been patched by attempting again to exploit the server, but this time attempting to redirect control flow to an invalid address. After this exploit attempt, we again make a request to the web server with a browser and verify that NGINX has withstood crashing (we configured NGINX to use a single worker, so a crash in a single worker would have resulted in the entire server being inoperable).

7 LIMITATIONS

One primary limitation of Piston is that the fully automated recovery steps only succeed on stack-based buffer overflows. For other types of corruption, an analyst typically needs to examine the data which was identified as corrupted, and then decide how it can be recovered. The reason for this limitation is that although the Data Filtering and Data Recovery in Section 4.3 can be thought of in generalized steps, they do not produce adequate results when applied to data outside of the stack.

Data Filtering. On the stack frame, we have the advantage of detecting which instructions access stack variables, whereas for data in the heap, due to limitations in the current state of static analysis, it is rare to know which instructions will read or write from a specific object. Some thorough type analyses [26, 36] may be able to identify accesses to objects of the same type, but cannot identify if those accesses are to the same object which was corrupted. Data filtering of heap corruption might require a *semantic* understanding of the program. Such understanding is outside the reach of current techniques.

Data Recovery. Data recovery requires data redundancy. That is, we must be able to automatically deduce the value of data from other values in memory or registers. In the case of stack data, we showed how other values can provide this redundancy in Section 4.3. However, if we consider corruption to heap or globals, one problem is that the data is typically created at an earlier point in program execution, often in a stack frame which has since been discarded. Unless we still have the stack frame in which a heap object was initialized, we are unlikely to have data which provides the necessary redundancy to recover the object.

However, there *are* cases when Piston can be applied to vulnerabilities other than stack overflows.

Here, we describe one such case, in which Piston was able to automatically patch a binary using a heap overflow vulnerability. The

Binary Name	**Function Interrupted?**	**Fully Automated Rollback?**	**Fully Automated Repair?**	**Caller Stack Bytes Recoverable**
CROMU_00017	Yes	Yes	Yes	144
CROMU_00020	Yes	Yes	Yes	52
CROMU_00037	No	N/A	Yes	4
CROMU_00038	Yes	Yes	No	4
CROMU_00039	Yes	Yes	Yes	303
NGINX	Yes	No	Yes	28

Table 1: Breakdown of patches from Piston

CGC binary NRFIN_00004, which was not included in our testing dataset because it does not have a stack-based buffer overflow, contains an intra-object heap overflow. The heap object contains a string followed by several function pointers. When the string overflows, the function pointers are overwritten, and another command handler will call an overwritten pointer.

We began by designing a custom heuristic to Piston to detect the corruption point. The heuristic was that for heap objects, any pointer to a function cannot be changed to point at an address that is not the beginning of a function. With this heuristic, Piston correctly identifies that the two function pointers in the heap object were corrupted. From there, Piston follows its normal mode of operation: it injects the patching stub into the binary, executes it, replaces all functions in the patch set, then restarts the execution of the hijacked function, which previously contained the heap overflow. Piston's underconstrained symbolic execution can detect that the corrupted pointers will be overwritten by the restarted (and patched) hijacked function, so no data needs to be recovered, avoiding the problem of the lack of data redundancy.

This is not a *general* application of Piston to heap overflows, so we include it here as opposed to the core approach discussion. However, it demonstrates that, with minor manual work, Piston can be adapted to a wider range of vulnerabilities. In this case, it only required a different corruption point detection heuristic.

8 RELATED WORK

Piston leverages many binary analysis techniques to analyze an executable, determine how to remotely apply the patch, exploit, and repair the remote process. In this section, we will detail work that proposed the program analysis techniques that we use in our system, and frame Piston in relation to other hot-patching techniques.

8.1 Hot-patching

Piston's core contribution is in extending the concept of hot patching to *remote* systems. This can include, like in our evaluation, remote user-space processes but, additionally, could include internet-connected embedded devices that may otherwise not have an update functionality.

Before Piston, hot patching techniques, or *dynamic software updating* approaches have been constrained to patching local processes, often with explicit support from the host system. Originally designed to patch small C programs, they have scaled up to the ability to patch the Linux kernel [5, 34, 35, 45, 46]. However, aside from being reliant on source code, these approaches require administrative access to the host machine, which is often unavailable.

To reduce the difficulty of and level of access required by hot-patching systems, techniques have been developed to include hot-patching support in the applications themselves. These systems, which are available for both user-space software [23, 24] and embedded device firmware [21, 27, 28], ease the administrative requirement, but still require pre-planning to include this functionality.

One hot-patching system, ClearView [38], is worth mentioning as it works by monitoring binary code, detecting when it is being exploited, and automatically generating and applying defensive patches. In the latter step, ClearView attempts to *repair* the state of the exploited process state by enforcing invariants. The concept of repairing the process state after exploitation is similar between ClearView and Piston. However, unlike ClearView, Piston does not require administrative access or, in fact, any presence on the device on which the process that needs patching is running. Piston patches, repairs, and resumes remotely, leveraging an exploit to achieve access.

Like most hot-patching systems, Piston relies on the analyst to provide a state transition routine when a patch that it is applying would modify data structures in the program. Recently, some work has been done in automatically recovering such a state transition routine [14, 20]. Though current work requires access to source code (which Piston does not have), a future extension of these techniques to binary code would increase the range of patches that Piston can automatically apply.

Exploit writers targeting operating system kernels have also found themselves repairing state of various parts of memory to allow the kernel to continue running after their exploit payload has been run. This is similar to the recovery and rollback routines used by Piston, but kernel exploiters have done this in a manual, ad hoc manner [39].

8.2 Code Injection

Piston patches binaries by injecting new code into the running process. The concept of injecting code at runtime with an exploit has been explored before, albeit not for patching purposes.

Windows malware often achieves code injection by inserting a DLL into the memory of the victim process [51]. This is done to add malicious functionality to a local process. However, this is done *locally*, as opposed to Piston's *remote* code injection, and cannot be done through an exploit. To our knowledge, Piston is the first approach that can inject its code *remotely*, via an exploit, and repair the damage caused by that exploit so that the application can continue.

8.3 Analysis Techniques

We utilize many existing binary analysis techniques to build Piston. However, we claim no advancement in the base of binary analysis:

Piston's contribution is in the application of binary analyses to remote hot-patching, in composing known analysis techniques in a novel way.

First, we use binary diffing techniques to identify what needs to be updated between the original and the replacement binary. This field has been extensively researched, and many approaches exist for identifying differences in executables, both statically [7, 8, 17, 19] and dynamically [18]. While we leverage diffing to determine what patches to apply to the remote process, diffing has also been used for everything from bug searching [40] to automatic exploit generation [9].

Once it determines the patches to apply, Piston uses program analysis techniques to create its repair plan. This includes a type of symbolic execution called under-constrained symbolic execution [41], which extends classical dynamic symbolic execution techniques [10, 12, 15] to work on isolated functions in a program. Additionally, we use static analysis techniques to recover the control flow graph of individual functions and to reason about data dependencies between stack variables. We leverage an open-source binary analysis framework, angr[2] [44] for this, which, in turn, uses several static analyses to recover control flow [16, 25, 42, 48, 52], identify variables [26], and determine data dependencies [6, 33, 47].

9 CONCLUSION

In this paper, we presented Piston, the first proposed approach for remote hot-patching of uncooperative processes. Piston patches processes through exploitation, allowing us to patch software which was originally considered unpatchable. Piston makes the novel contribution of exploitation clean up, recovering from many of the unpredictable state changes introduced during a memory corruption exploit. We evaluated Piston on a large, real-world binary and a synthetic dataset provided by DARPA. Piston was able to apply patches to each binary and, in most cases, carried out the patch completely automatically.

ACKNOWLEDGMENTS

This material is based upon work supported by ONR under Award Numbers N00014-17-1-2897, N00014-17-1-2011, N00014-15-1-2948. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the ONR.

REFERENCES

[1] Cve-2013-2028 advisory. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-2028.

[2] Nginx cve 2013-2028 kingcope exploit. https://www.exploit-db.com/exploits/26737/.

[3] Nginx cve 2013-2028 patch. http://nginx.org/download/patch.2013.chunked.txt.

[4] Vulnerability distribution of CVE security vulnerabilities by type. http://www.cvedetails.com/vulnerabilities-by-types.php.

[5] J. Arnold and M. F. Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 187–198. ACM, 2009.

[6] G. Balakrishnan and T. Reps. WYSINWYX: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(6):23, 2010.

[7] M. Bourquin, A. King, and E. Robbins. Accurate comparison of binary executables. 2013.

[8] M. Bourquin, A. King, and E. Robbins. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, page 4. ACM, 2013.

[9] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 143–157. IEEE, 2008.

[10] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[11] F. Castaneda, E. C. Sezer, and J. Xu. Worm vs. worm: preliminary study of an active counter-attack mechanism. In *Proceedings of the 2004 ACM workshop on Rapid malcode*, pages 83–93. ACM, 2004.

[12] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 380–394. IEEE, 2012.

[13] Y.-Y. Chang, P. Zavarsky, R. Ruhl, and D. Lindskog. Trend analysis of the cve for software vulnerability management. In *Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on*, pages 1290–1293. IEEE, 2011.

[14] H. Chen, J. Yu, C. Hang, B. Zang, and P.-C. Yew. Dynamic software updating using a relaxed consistency model. *Software Engineering, IEEE Transactions on*, 37(5):679–694, 2011.

[15] V. Chipounov, V. Kuznetsov, and G. Candea. *S2E: A platform for in-vivo multi-path analysis of software systems*, volume 47. ACM, 2012.

[16] C. Cifuentes and M. Van Emmerik. Recovery of jump table case statements from binary code. In *Program Comprehension, 1999. Proceedings. Seventh International Workshop on*, pages 192–199. IEEE, 1999.

[17] T. Dullien and R. Rolles. Graph-based comparison of executable objects (english version). *SSTIC*, 5:1–3, 2005.

[18] M. Egele, M. Woo, P. Chapman, and D. Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 303–317, 2014.

[19] H. Flake. Structural comparison of executable objects. 2004.

[20] C. Giuffrida, C. Iorgulescu, A. Kuijsten, and A. S. Tanenbaum. Back to the future: Fault-tolerant live update with time-traveling state transfer. In *LISA*, pages 89–104, 2013.

[21] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Safe and automatic live update for operating systems. *ACM SIGPLAN Notices*, 48(4):279–292, 2013.

[22] D. Goodin. Windows 7, not xp, was the reason last weekâĂŹs wcry worm spread so widely, 2017. https://arstechnica.com/security/2017/05/windows-7-not-xp-was-the-reason-last-weeks-wcry-worm-spread-so-widely/.

[23] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, and J. S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for c. In *ACM SIGPLAN Notices*, volume 47, pages 249–264. ACM, 2012.

[24] C. M. Hayden, E. K. Smith, M. Hicks, and J. S. Foster. State transfer for clear and efficient runtime updates. In *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, pages 179–184. IEEE, 2011.

[25] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *USENIX security Symposium*, volume 13, pages 18–18, 2004.

[26] J. Lee, T. Avgerinos, and D. Brumley. TIE: principled reverse engineering of types in binary programs. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011.

[27] H. Martorell, J.-C. Fabre, M. Roy, and R. Valentin. Towards dynamic updates in autosar. In *SAFECOMP 2013-Workshop CARS (2nd Workshop on Critical Automotive applications: Robustness & Safety) of the 32nd International Conference on Computer Safety, Reliability and Security*, page NA, 2013.

[28] H. Martorell, J.-C. Fabre, M. Roy, and R. Valentin. Improving adaptiveness of autosar embedded applications. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 384–390. ACM, 2014.

[29] G. McManus, hal, and saelo. Nginx cve 2013-2028 metasploit exploit. https://github.com/rapid7/metasploit-framework/blob/master/modules/exploits/linux/http/nginx_chunked_size.rb.

[30] M. A. McQueen, T. A. McQueen, W. F. Boyer, and M. R. Chaffin. Empirical estimates and observations of 0day vulnerabilities. In *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*, pages 1–12. IEEE, 2009.

[31] C. Mulliner, J. Oberheide, W. Robertson, and E. Kirda. Patchdroid: Scalable third-party security patches for android devices. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 259–268. ACM, 2013.

[32] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras. The attack of the clones: a study of the impact of shared code on vulnerability patching. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 692–708. IEEE, 2015.

[33] J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Signedness-agnostic program analysis: Precise integer bounds for low-level code. In *Programming Languages and Systems*, pages 115–130. Springer, 2012.

[34] I. Neamtiu and M. Hicks. Safe and timely updates to multi-threaded programs. In *ACM Sigplan Notices*, volume 44, pages 13–24. ACM, 2009.

[35] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. *Practical dynamic software updating for C*, volume 41. ACM, 2006.

[2] Available at `https://github.com/angr/angr`

[36] M. Noonan, A. Loginov, and D. Cok. Polymorphic type inference for machine code. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 27–41. ACM, 2016.

[37] Oracle. Ksplice. http://www.ksplice.com/.

[38] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, et al. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102. ACM, 2009.

[39] E. Perla and M. Oldani. *A guide to kernel exploitation: attacking the core*. Elsevier, 2010.

[40] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 709–724. IEEE, 2015.

[41] D. A. Ramos and D. Engler. Under-constrained symbolic execution: correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, 2015.

[42] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Reverse engineering, 2002. Proceedings. Ninth working conference on*, pages 45–54. IEEE, 2002.

[43] Secunia. Resources vulnerability review 2015. http://secunia.com/resources/vulnerability-review/introduction/.

[44] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. 2015.

[45] M. Siniavine and A. Goel. Seamless kernel updates. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2013.

[46] A. Sotirov. Hotpatching and the rise of third-party patches. BlackHat USA, 2006.

[47] Tok, Teck Bok and Guyer, Samuel Z and Lin, Calvin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In *Compiler Construction*, pages 17–31. Springer, 2006.

[48] J. Troger and C. Cifuentes. Analysis of virtual method invocation for binary translation. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 65–74. IEEE, 2002.

[49] R. van der Meulen. Gartner says 6.4 billion connected "things" will be in use in 2016, up 30 percent from 2015. http://www.gartner.com/newsroom/id/3165317.

[50] R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Steal this movie: Automatically bypassing drm protection in streaming media services. In *USENIX Security*, pages 687–702, 2013.

[51] Wikipedia. Dll injection. https://en.wikipedia.org/wiki/DLL_injection.

[52] L. Xu, F. Sun, and Z. Su. Constructing precise control flow graphs from binaries. *University of California, Davis, Tech. Rep*, 2009.

A APPENDIX

A.1 Example

To shed more light into Piston's operation, we provide an example binary in which Piston can automatically patch out a stack-based buffer overflow. In Listing 2 there is an overflow in line 12 with the call to `gets()`. Piston will receive as inputs the original binary for the code in Listing 2, the patched version, and an exploit specification to achieve code execution. Note that we show the source code for clarity, although Piston will run entirely on the compiled executables.

Piston will execute the following high-level steps to remotely patch a process running the code in Listing 2:

Patch generation. Using binary diffing techniques, Piston will identify that the `hello()` function has changed in the replacement binary. Piston then prepares a patch to insert the updated `hello()` function into the memory of the remote process.

Repair planning. Piston will analyze the exploit specification and trace the exploit off-line to determine what is corrupted during the exploit. As shown in Figure 2 (b) the return address and value of the variable `counter` are corrupted. Piston will automatically generate a repair routine to recover the value of the stack variable `counter` as well as the return address. In Figure 2 (c) the arrow between `main_counter` and `counter` shows that the repair plan uses the value of `main_counter` to restore the value of `counter`.

Remote patching. Piston will exploit the process using the specification provided to inject the patcher core. The patcher core first receives the repair routine from Piston, which is used to restore the corrupted stack values. Then the patcher core receives the prepared patches to replace the `hello()` function. Finally, the process execution will be resumed on line 10, at the start of the patched `hello`, now running a patched version of the function.

```
 1 void main() {
 2   int main_counter = 0;
 3   while(1) {
 4     hello(main_counter);
 5     main_counter += 1;
 6     ...
 7   }
 8 }
 9 void hello(int counter) {
10   char buf[0x20];
11   puts("Enter your name:\n");
12   gets(buf);
13   printf("hello
         %s, you are vistor %d\n", buf, counter);
14   ...
15 }
```

Listing 2: An example of a stack-based overflow that Piston can patch.

A.2 Glossary

We have been careful to use consistent terminology throughout the paper, and collect its definitions here.

Corruption effects. A register or memory write that is influenced by data that was corrupted by the exploit.

Exploitation point. The point, in the exploitation trace, at which it becomes apparent that the binary has been exploited.

Exploit specification. Fundamentally, a script that carries out an exploit against the remote process to achieve remote code execution.

Exploitation trace. A detailed trace of the exploit running against the original binary (configured with the remote configuration). This trace is analyzed to generate the rollback and repair functions.

Hijacked function. The function from which the exploit hijacks control flow.

Original binary. The binary that is currently running on the remote process.

Overflow instruction. The instruction that performs the write that triggers the detection of the exploitation point.

Patch set. The specific set of patches that Piston will apply to the remote process to transform it into a functional copy of the replacement binary.

Patching stub. A small payload that is injected by Piston into the remote process to facilitate the various patching tasks.

Persistence routine. An optional routine, provided by the user, that tries to persist Piston's changes on the remote machine (i.e., by overwriting the original binary with the replacement binary). In most use-cases for Piston, this is not actually possible due to lack of access.

Recovery set. A set of registers and stack variables that Piston has determined need to be recovered before resuming remote process execution.

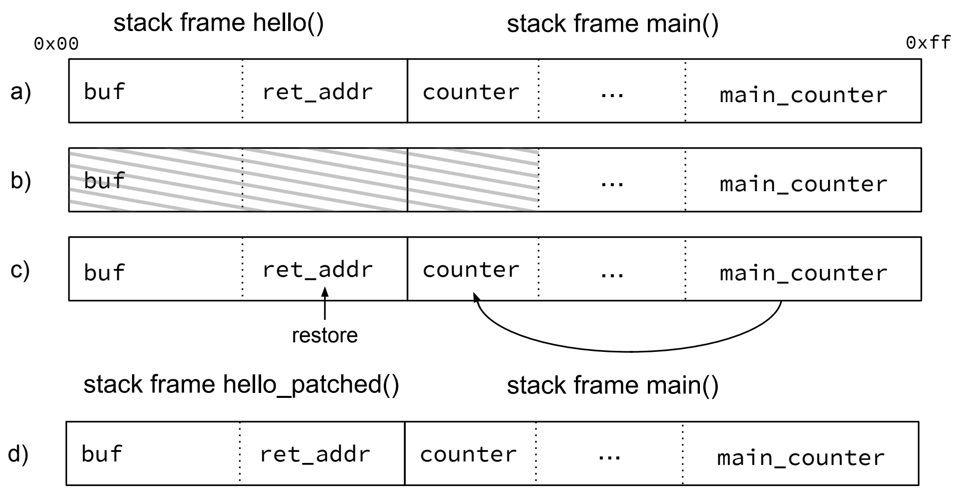

Figure 2: A view of the stack frames of the program shown in Listing 2 during the automated repair process. (a) Here the program is currently executing the function `hello()`. At the bottom of the stack frame of `hello()` is the return address, followed by the variable `counter`. (b) After the call to `gets()` the buffer overflows, corrupting the values of `ret_addr` and `counter`. (c) Piston restores the value of `counter` using the redundant data on the stack that was not corrupted, specifically the value of `main_counter`. The value of the return address is also restored. (d) Piston replaces `hello()` with the new version of the function `hello_patched()` which is taken from the patched binary. Finally, Piston chooses to restart the function `hello_patched()` and program execution continues safely.

Remote configuration. The configuration of the remote process. For example, if the remote process is a web server, this would be the configuration file of the web server. Piston uses this to recreate an accurate exploit trace.

Remote process. A process (or piece of firmware) running on the remote system that the analyst wants to patch.

Repair routine. A function, either generated by Piston or provided by the analyst, that repairs state corruption in the remote process after Piston exploits it.

Replacement binary. The binary that the analyst wants to replace the original binary with on the remote process.

Rollback routine. A function, either generated by Piston or provided by the analyst, that *undoes* the effects of the hijacked function when the exploit prevents the hijacked function from completing properly.

Scratch Space. Memory that can be used inside of a function, but will not be accessed outside of the function. This includes local stack space as well as data in the heap that will be freed before the end of the function.

State transition routine. A function provided by the analyst which is necessary when a patch introduces changes to structures. This function is responsible for updating all instances of the affected structures to fit the new definition of the structure.