Formal Methods in Computer-Aided Design 2020

Using model checking tools to triage the severity of security bugs in the Xen hypervisor

Byron Cook[*][†], Björn Döbel[*], Daniel Kroening[*][‡], Norbert Manthey[*],
Martin Pohlack[*], Elizabeth Polgreen[§][||], Michael Tautschnig[*][¶], Pawel Wieczorkiewicz[*]
[*]Amazon Web Services [†]University College London [‡]University of Oxford
[§]UC Berkeley [¶]Queen Mary University of London
[||]University of Edinburgh

***Abstract*—In practice, few security bugs found in source code are urgent, but quickly identifying which ones are is hard. We describe the application of bounded model checking to triaging reported issues quickly at the cloud service provider Amazon Web Services (AWS). We focus on the job of reactive security experts who need to determine the severity of bugs found in the Xen hypervisor. We show that, using our publicly available extensions to the model checker CBMC, a security expert can obtain traces to construct security tests and estimate the severity of the reported finding within 15 minutes. We believe that the changes made to the model checker, as well as the methodology for using tools in this scenario, will generalise to other organisations and environments.**

I. Introduction

Some bugs have serious security implications. For well-engineered systems most bugs do not. The reason is that these systems are built with *defense in depth* [1], meaning that the average bug found usually just temporarily reduces the depth of the defense provided until the bug is fixed, but does not present an immediate security concern that nullifies assumed defenses.

At Amazon Web Services (AWS), a key challenge we face is quickly categorising each bug report, which requires answering the question whether the bug is reachable through all security layers. Determining vulnerability severity is performed under intense time-pressure, as Amazon's first priority is the continuous security of its customers. The key to success in these situations is access to quick and accurate answers to questions about state-space reachability. Our case study focuses on determining the severity of bugs in Xen [2], an open-source hypervisor used throughout the industry. AWS uses a customised Xen version on some of its Elastic Compute Cloud (EC2) servers. While this case study focuses on Xen, we believe the results generalise to other large-scale systems: based on our experience with Xen and other systems, after an initial effort to ensure successful builds of the code base, the system-specific effort to apply the approach reported on in this case study is low.

For each security finding, the Xen Project publishes a *Xen Security Advisory* (XSA) [3]. A typical XSA comes with a description of the problem and a source-code patch to mitigate the issue. Before full publication, the XSA is shared with the members of Xen's pre-disclosure list, as is common in responsible disclosure processes. At AWS, members of 24/7 security operations triage potential security bugs as they are discovered or reported. They may find themselves in the following quandary: should they wake the engineering team from their beds to investigate? Or do the existing layers of defense mitigate against the consequences of the bug? Often the same code is used in multiple products, where the defenses will differ from service to service.

In order to assess the severity of a given XSA, the security expert will manually determine whether the vulnerability is reachable in the AWS-customised version of Xen. Engineers construct *security tests* to reproduce the vulnerability, thereby answering the reachability question. This reachability question fundamentally is a global question about the interaction amongst details across the entire EC2 system. It includes complex custom hardware, software, protocols, and networks that implement the layers of security defense, as well as enabling high compute utilization and scalability needed for one of the world's largest cloud providers.

In this case study we describe our use of bounded model checking to help our security experts make faster and more accurate assessments of severity. The complexity of the overall environment is well beyond the capacity of today's formal methods tools. We automate the part of the process that was previously most time-consuming for security experts using an extended version of CBMC [4]. For a given XSA, we use the source-code patch provided with the XSA to write an assertion, which we insert into the patched source-code area. Reachability and violation of this assertion indicates a possible exploitation of the vulnerability. We analyse the Xen source code in the context of a potential security bug and generate traces that are helpful for test construction.

These tests will be executed in the overall EC2 environment, and help the security experts among us understand which defenses remain intact, or else help find a complete proof-of-concept test, which will be used to confirm any mitigation. In our experience, we can perform work in minutes that would previously have taken weeks or months. Because weeks is unacceptably long, before the use of our methodology, additional developers were enlisted as needed in order to reach a more timely analysis conclusion. Now, we are able to more rapidly make high-confidence calls using the high-fidelity answers produced by our methodology. The result is that the rare critical security bugs get fixed even faster than before, with fewer human resources.

https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_26

Related work: We integrate our extensions into CBMC [5], [6], a bit-precise bounded model checking tool for C programs. Model checking is frequently applied to security problems: Gallagher et al. [7] use security patches to generate verification assertions, and use CBMC and Frama-C to verify these. UQBTng [8] automatically finds integer overflows in Win32 binaries using CBMC. Vasudevan et al. [9] use CBMC in the verification of a small hypervisor framework. Automated verification techniques have been applied to the address translation subsystem [10] of the Xen hypervisor, using a parametric verification technique to reduce the model size. A small custom hypervisor is analysed by Alkassar [11] and [12]. Dahlin et al. [13] develop a simple but fully verified hypervisor. None of these approaches scales to the size of Xen (cf. Section III).

Frama-C has been applied to verify a subset of the Xen hypervisor code [14], using modelling of assembly code, harness functions and manually picking hypercalls. The considered properties are shallower than required for analysing security issues. In contrast, we aim to automatically select hypercalls, and focus on the interaction between the guest and the hypervisor. KLEE [15] cannot be applied, as Xen does not compile to LLVM [16]. This rules out tools built on top of KLEE, such as the automatic exploit generation tool of [17], and also the concolic execution approach that Chen et al. presented for Linux kernel modules [18].

II. CBMC and Bounded Model Checking

CBMC [6] is a bounded model checker [19] which can check for the violations of assertions in C programs, or prove absence of violations given a specific bound. The default behaviour of CBMC is that it unwinds any loops or recursion in the program to a given unwinding limit but unrolls the rest of the program fully. CBMC performs a bit-precise translation of the source program into a Boolean formula, which is then passed to a SAT solver. If the formula is satisfiable, then a counterexample trace that leads to a violated assertion exists, and CBMC translates the model returned from the SAT solver back into assignments to program state variables for each state in this trace.

CBMC's code base also provides for a number of text-book data-flow analyses. These include precise slicers as discussed in Section III-B. In Section IV we explain as to why we had to add approximate slicing to complement the precise ones.

III. The Xen Hypervisor

In cloud computing, one physical *host machine* is partitioned into several parts, called *virtual machines*. Virtual machines behave like complete computers with their own operating system, and each virtual machine serves a single *guest* (serving as a host in nested virtualisation [20]). The software that provides this illusion is called a hypervisor [21], [22]. Xen [2] is a bare-metal hypervisor, as it runs directly on the hardware of the host and manages all the host's resources. A similar hypervisor is KVM [23], for which this work would apply as well.

1) Virtual machines: Every guest virtual machine in Xen has its own guest kernel and operating system. A system call from a program of a guest, e.g., a request for access to I/O devices, reaches the guest kernel. In most configurations of Xen, the guest kernel does not have direct control of the physical machine. Hence, the guest kernel issues a *hypercall* or accesses a predefined memory range to request the service from the hypervisor. Xen is event-driven: after booting and once a guest runs, Xen waits for guest code to execute and takes actions on hypercalls, or host interrupts. The hypervisor handles hardware exceptions and interrupts, which may be raised by the CPU when guests issue privileged instructions.

2) Memory: Xen uses *virtual memory* for isolation and to give guests the impression they are working with contiguous sections of memory, when the physical memory could be spread across different locations. Virtual memory is split into fixed-length contiguous blocks called *pages* and each *virtual address*, describing a location in a page, is mapped to a physical address in a *page frame*. This mapping is stored in a *page table*. There are several ways Xen can virtualise memory; most commonly Xen uses hardware support in the form of nested paging and extended page tables.

A. Example Vulnerability and Security Test

In the presence of an adversarial guest, security issues can result in information leaks, guest denial of service (DoS), privilege escalation to or DoS of the host machine. We discuss XSA 227 [24] as an example of a potential vulnerability.

1) Security vulnerablity XSA 227: A guest can share memory with, e.g., other guests, or devices. When setting up a new shared memory area, the guest passes the memory location to be shared to the hypervisor, by giving the *guest-physical address* of an entry in one of its page tables. To share the page, the hypervisor modifies this page table entry. Before the modification, Xen checks several properties. XSA 227 reported that Xen did not check whether the entry address starts at the beginning of a page table entry, i.e., whether the address is *aligned*. Since Xen writes exactly one page table entry, the hypervisor can write beyond an unaligned page table entry, allowing the guest to partially overwrite the next page table entry. Writing to the page table in this unprotected way is sufficient to allow a guest to grant itself additional permissions and gain full system access.

2) Security test for XSA 227: To establish the severity of an XSA for AWS, an engineer develops tests to trigger the vulnerability in the EC2 environment. For XSA 227, the test performs a hypercall from a guest that shares memory at a non-aligned address. If the hypercall returns with success, the vulnerability is reachable. Else, if the hypercall returns an error code—and if no other mistakes have been made when invoking the hypercall—the vulnerability is unreachable.

3) Equivalent reachability problem: For XSA 227, part of the XSA patch provided is the following macro that checks whether the page-table entry of a page is aligned:

```
#define IS_ALIGNED(val, align) \
  (((val) & ((align) - 1)) == 0)
```

We use this macro to add an assertion that, if violated, indicates the vulnerability is exploitable. For XSA 227, we insert the following `assert` statement over local variables `pte_addr` and `nl1e` into the source-code area patched in the XSA:

```
assert(IS_ALIGNED(pte_addr, sizeof(nl1e)));
```

We can thus use software model checking to verify whether the above assertion can be violated, starting from the hypercall entry point. Such a counterexample can be used to construct the above mentioned test.

B. Challenges in Applying Automated Program Analyses

Existing program analysis techniques, however, cannot be applied out-of-the-box to the Xen code base: (1) Xen uses C code with systems extensions and assembly code throughout the codebase, for example interfacing with hardware. In Xen 4.8, there remain more than 700 lines of assembly in the code base. To the best of our knowledge, there is no symbolic verification tool that can handle this combination of C and assembly code on such a large code base. (2) Modelling behaviour of an adversarial guest precisely would require modelling the exact start state of the machine, which is determined in boot code, the exact interaction history with other guests, and maintaining a full model of the memory layout and system registers. We cannot do this, due to both the proliferation of low-level assembly code in the boot code and the scalability demands of a full memory model. (3) Xen is configurable, due to its requirement to have full control of a machine regardless of architecture. Thus, Xen contains code that emulates CPU instructions for multiple different architecture flavours. During boot time, the architecture flavour is determined and function pointers are set to point to the correct implementations. (4) The size of the code base exceeds the scalability limits of existing software model checkers: Xen 4.8 is comprised of ∼300,000 lines of code. Benchmarks in the TACAS Competition on Software Verification [25] are smaller. The largest tasks feature ∼100,000 lines of code. However, in 2019, large solved benchmarks in this competition typically had either short counterexample traces or simple proofs of safety. For Xen, we expect long counterexample traces of instructions to refute safety, due to the steps involved in the interaction with hypervisors. For all XSAs we have investigated, unpatched CBMC failed to complete analysis within an 8 hour time window, even after using all available program slicers. Program slicing [26] uses dependence analysis to remove instructions that cannot affect a property of interest. CBMC includes a reachability slicer, which removes instructions that cannot affect any assertion, and a slicer to remove code that initialises unused global variables. These slicers are fast but do not remove enough code for the analysis of Xen to become feasible. CBMC also contains a full-program slicer, which computes the cone-of-influence [27]. Full slicing is precise but, owing to the cost of points-to analyses, not scalable and does not complete within 8 hours.

IV. Extending CBMC to handle Xen

We address these four challenges using automated approximations with hooks for expert-provided refinement, implemented in an extended version of CBMC [4].

```
void do_hypercall()
{
  int nondet;
  switch(nondet)
  {
  case 1:
    XEN_GUEST_HANDLE (const_trap_info_t) traps1;
    do_set_trap_table(traps1);
    break;
  case 2:
    XEN_GUEST_HANDLE (mmu_update_t) ureqs2;
    unsigned int count2;
    XEN_GUEST_HANDLE (uint) pdone2;
    unsigned int foreigndom2;
    do_mmu_update(ureqs2, count2, pdone2, foreigndom2);
    break;
  case 3:
    XEN_GUEST_HANDLE (ulong) frame_list3;
    unsigned int entries3;
    do_set_gdt(frame_list3, entries3);
  ...
```

Figure 1. Model of the hypercall table. This is modelled as a non-deterministic switch over all possible hypercalls, called with non-deterministic arguments.

A. Assembly Code

When the lack of interpretation of assembly code adversely affects precision, we model it in C, most importantly the hypercall table of Xen, which contains the hypercalls a guest may use. We model this as a non-deterministic choice over hypercalls, entered with non-deterministic arguments, to allow all possible guest behaviour. Figure 1 shows a snippet from this model. While this model is currently constructed manually, it is reused across XSAs. Future work on CBMC includes adding native support for assembly code, which will avoid the need for expert-provided input for this stage.

B. Environment Modelling

We start our analysis either at a start point known to be relevant to the XSA or at the hypercall entry-point. To over-approximate the state of the machine at the point the guest makes a hypercall, we automatically generate an environment that assumes non-deterministic values for all input parameters to the start function, and constrains all pointers to refer to valid areas of memory. To enforce the latter, we add harness functions into the code at analysis time, as shown in Figure 3. These functions initialise the pointers to point to valid but non-deterministic objects. By starting from this set of states, we over-approximate the potential behaviour of the adversary between boot and the first hypercall we model.

C. Function Pointer Removal

By default, CBMC expands function pointers to a case statement over a set of functions determined using an over-approximating signature-based analysis. For the configurable code base of Xen, the signature-based analysis yields up to 300 functions for a single function pointer. CBMC determines the precise set of function calls, i.e., the subset of feasible cases,

```
#define ARGS(x, n)                    \
    [__HYPERVISOR_ ## x ]={n, n}
#define COMP(x, n, c)                 \
    [__HYPERVISOR_ ## x ]={n, c}

const hypercall_args_t
  hypercall_args_table[NR_hypercalls] =
{
    ARGS(set_trap_table, 1),
    ARGS(mmu_update, 4),
    ARGS(set_gdt, 2),
  ...

#define HYPERCALL(x)                  \
    [ __HYPERVISOR_ ## x ] =          \
      { (hypercall_fn_t *) do_ ## x,  \
        (hypercall_fn_t *) do_ ## x }
#define COMPAT_CALL(x)                \
    [ __HYPERVISOR_ ## x ] =          \
        {(hypercall_fn_t *) do_ ## x, \
        (hypercall_fn_t *) compat_ ## x }
  ...

static const hypercall_table_t
    pv_hypercall_table[] = {
    COMPAT_CALL(set_trap_table),
    HYPERCALL(mmu_update),
    COMPAT_CALL(set_gdt),
  ...
```

Figure 2. Fragments of the original code that builds the hypercall table

during symbolic execution. As our analysis running on Xen uses non-deterministic initial states, symbolic execution would typically deem *all* cases feasible, even though most of them are spurious. To reduce the candidate set per pointer, we now use CBMC's existing flow-insensitive points-to analysis [28] in place of the signature-based analysis. This flow-insensitive analysis is field-sensitive, therefore the analysis distinguishes the fields of every object, while merging the values of them for different program locations. If the flow-insensitive points-to analysis yields an empty set as, e.g., caused by pointers depending on boot code, we fall back to the original behaviour. With this change, we introduce 20k fewer function calls, about 114k instead of 134k, and hence reduce the likelihood of spurious counterexamples.

Configurable harnesses: Xen code supports several architectures. For our analysis we pick a single set of architectures, i.e., Intel 64bit CPUs. This allows us to restrict the set of functions considered for handling architecture specifics, while still using a non-deterministic machine state. We thus support adding expert-provided code into the harnesses described in Section IV-B to restrict the candidates of these function pointers to a specific function or a non-deterministic choice over a constrained set of functions, e.g., excluding all AMD-specific functions.

The example given in Figure 3, `x86_emulate` contains several function pointers and an expert engineer specifies two possible function pointers for the read function.

D. Approximating Program Slicer

In order to focus analysis on the relevant part of the hypervisor, we introduce a more aggressive slicing approach

```
int main()
{
  struct x86_emulate_ctxt harness_ctxt;
  struct x86_emulate_ops harness_ops;
  int nondet;
  // instantiate read function pointer
  switch(nondet)
  {
  case 1:
    harness_ops.read = EXAMPLE;
    break;
  case 2:
    harness_ops.read = EXAMPLE2;
    break;
  }
  // expert restricts possible vendor values
  __CPROVER_assume(harness_ctxt.vendor < 3);

  x86_emulate(&harness_ctxt, &harness_ops);
}
```

Figure 3. Harness for `x86_emulate`. An expert can provide restrictions over the input space, such as the restriction on the values of vendor variable shown in the assumption here. Uninitialised variables such as `nondet` are considered free variables to be assigned any non-deterministic value by the underlying SAT solver.

following the algorithm of Figure 4: we first compute an approximation of the call graph using the function-pointer removal as described above. Using this call graph, the slicer computes the set of paths from the entry point to the target property. From this set, we select *direct paths*, which we define as the paths without cycles on the call graph. We then take these direct paths and remove all function calls that return back to the calling function and replace these function calls with an approximation of their behaviour by *havocking* the function body, as explained in Section IV-D1. This produces an approximation of the cone of influence, which may be sufficiently small to analyse with CBMC.

If the resulting program slice is still too large to analyse, the approximating slicer can be configured to keep only functions on the shortest direct path. To increase precision, we can preserve all functions within a given distance of function calls from the direct paths, illustrated in Figure 5. To obtain scalability and precision, we run multiple analyses with varying degrees of precision in parallel to find the configuration that completes within the timeout with maximum precision.

1) Approximating behaviour of missing code: If a function is removed, we must approximate the behaviour of that function in order to avoid missing counterexamples. A coarse approximation is to *havoc* the function, i.e., assume the function may return a non-deterministic value and may assign a non-deterministic value to any arguments passed by pointer. This approximation is not strictly an over-approximation, because it may under-approximate behaviour as described below. We chose this simplification, because computing and refining a sound over-approximation is computationally intensive, and missing some counterexamples due to under-approximation is acceptable for our use case as we strive to support the security expert in constructing tests.

```
    Approximating_Slice (CFG $g$, node entry, node target, bool direct, int distance)
s1  $FP$ := remove_function_pointers($g$)
s2  $CG$ := compute_call_graph($FP$)
s3  $DP$ := get_direct_paths($CG$, entry, target)
s4  $DP$ := shortest_path($DP$) **if** $\neg$ direct **else** $DP$
s5  mark_for_havoc $= \emptyset$
s6  **for** node $n$ **in** $FP$:
s7    **if** distance($FP$, $DP$, $n$) $>$ distance:
s8      mark_for_havoc := mark_for_havoc $\cup \{n\}$
s9  **for** node $n$ **in** mark_for_havoc:
s10   havoc_object($n$)
```

Figure 4. Approximating slicing is applied to input program represented by its control-flow graph g, and configurable in the entry- and target nodes, whether or not to consider all direct paths, and the maximum distance.

2) Potential under-approximation: The first source of under-approximation are global variables written to by a function that we removed. We partly mitigate the absence of modelling this behaviour by starting our analysis in a non-deterministic initial state, including non-deterministic global variables. It is, however, possible that a trace requires a global variable to take different values during the trace; such counterexamples would thus be missed.

The second source of under-approximation is not havocking pointers to pointers. When a function receives a pointer `A` as argument that points to pointer `B`, we do not havoc pointer `B`. When a function receives a pointer `A` that points to a **`struct`** `B` that contains a pointer `C`, we do not havoc pointer `C`. We choose not to havoc these pointers, as this can change any memory to any value, and introduces spurious counterexamples. There are 84 functions in Xen that accept pointers to pointers, and experts did not find any to be relevant to the XSAs we analysed.

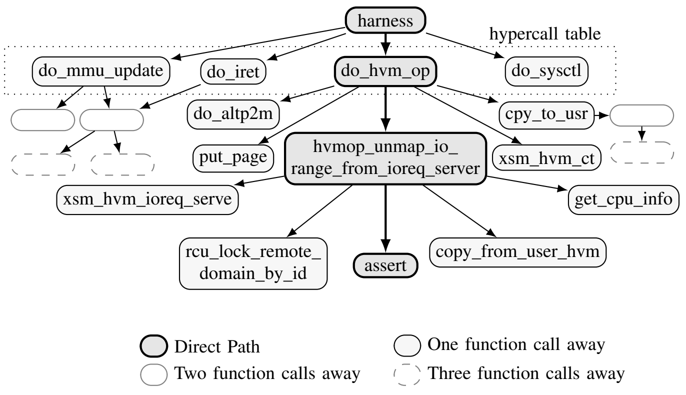

Figure 5. Xen's call graph from the harness function to an assertion representing XSA 238. The thick framed nodes show the *direct path*; these functions are always preserved by the approximating slicer. The thin, solid framed nodes show functions which will be approximated, as described in Section IV-D1, by default. If we preserve functions up to one function call away from the direct paths, the light grey nodes will be preserved, and the unlabelled nodes represent functions which will be approximated.

V. Determining Severity of Vulnerabilities

Our aim is to assist experts in determining the severity of a security vulnerability, within a specific version of Xen. To illustrate this use case, we selected a few XSAs with different properties: 200, 212, 213, 227, and 238 [3]. We build our modifications on top of CBMC version 5.10, which uses MiniSat 2.2.1 [29]. We disable MiniSAT's pre-processor, as it usually consumes more run time than the actual verification task for the given problems. We pick Xen release 4.8 [30], as none of the selected XSAs have been mitigated in this version, and fixed handling comments in assembly to allow us to compile the code with CBMC. Next, we added the assertions and harness functions for each XSA. This is a required setup step the complexity of which varies between adding a single assertion, and adding a full harness for the hypercall table (about 300 instructions), depending on the XSA. When starting from our provided package, these harnesses are already present, and hence, future harnesses require less effort. To speed-up overall time, and precision, we run multiple configurations of the slicer and analysis options in parallel via AWS Batch [31], to obtain first results quickly. Our Xen and CBMC packages including all scripting are available for download[1] and the aggressive slicer is available in the main CBMC branch[2].

The counterexample trace contains all function entries and exits, arguments and relevant variable assignments. We add an option to CBMC to print the trace in HTML with options to expand function calls.

A. Results

The experiments were run on AWS Batch using the EC2 r5 instance family, with a memory limit of 110 GiB and an overall timeout of 8 hours per job. The original Xen binary contains 103,662 effective program locations, i.e., code statements that affect the state of the program. We ran CBMC out-the-box on each of the XSAs with all combinations of the CBMC program slicers, with a loop unwinding limit of 0, i.e., executing the loop body just once. The reachability slicer and global slicer reduce the instruction count by up to 20 %. CBMC cannot

[1]https://github.com/nmanthey/xen/tree/FMCAD2020
[2]https://github.com/diffblue/cbmc

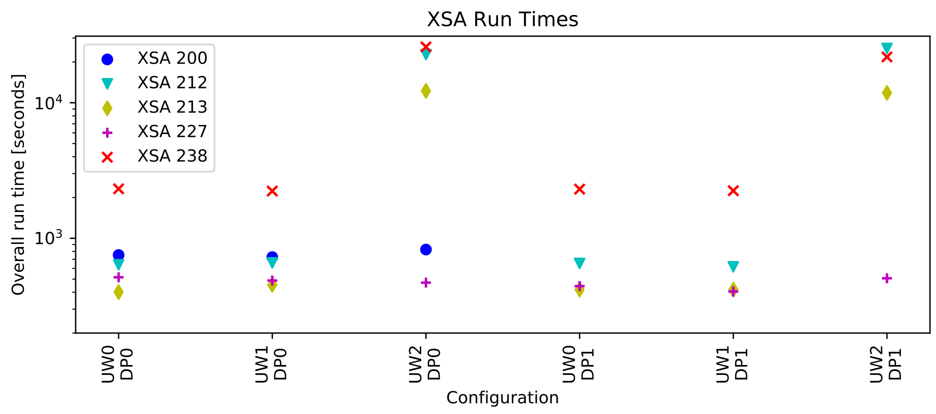

Figure 6. Run time of the overall approach for selected configurations that finish within 8 hours. We fixed the parameters to distance=2, and advanced function pointer removal as well as run full slicing after approximating slicing. Keeping all direct paths (DP1), as well as unwinding loops (UW) during search are altered.

produce a results for any of the combinations, and the full slicer does not finish within the 8 h timeout.

We vary the input parameters to approximating slicing (cf. Figure 4) to preserve all direct paths, or shortest paths only, or preserve functions with a distance of up to 2 calls. We limit loop unwinding to 0, 1 or 2 iterations, and use both function pointer removal approaches. These limits were chosen since they were large enough to produce precise enough results to create tests from the traces. It is possible to increase these limits and still obtain traces, but substantially large values may result in a binary too large to analyse in reasonable time (for instance, increasing the depth to 10 or greater).

Slicing is crucial, as it reduces the size of the input program to less than 5 % of its original size in under 10 minutes. Overall run times of the more precise configurations (distance=2) are presented in Figure 6. For smaller distances, the run times are typically smaller. The figure shows that run times depend on the XSA, as well as on the unwinding parameter – for more unwinding the run time for XSAs 213, 212 and 238 increases. For XSA 200, no direct-path based traces can be produced within 8 hours in case all direct paths are kept, because XSA 200 is located in instruction emulation code, which introduces many direct paths.

For all five selected XSAs, an initial result for at least one configuration is returned within 10 minutes. This time allows engineers to refine the harness to improve the result for test generation quickly. Within the first hour, more than 30 configurations produce traces.

B. Turning a Trace To a Test

To make the results of this work consumable for future XSAs, we also discuss how to turn a counterexample trace from CBMC into a security test that could be executed inside the guest. The required information is

1) the configuration of Xen
2) the type of the guest that can hit the security issue
3) the interaction the guest has to perform to trigger the security issue

A typical XSA description provides data for item 1 and 2, because it scopes the security issue. In case CBMC produces a valid trace for an assertion, this trace provides information about item 3, namely the relevant data that comes from the guest. This data is forwarded to Xen via a few interfaces:

1) hypercalls, namely the call to perform, as well as the arguments for the hypercall
2) `copy_from_guest`, a function which copies data from the guest into the hypervisor
3) hardware interaction, e.g., content of packages that are generated by interaction with the (emulated) hardware

In the XSAs we analysed, we only see interaction via hypercalls and the function `copy_from_guest`. No devices are involved in these XSAs.

Finally, to turn the relevant parts of the trace into an actual security test, we need the basic building blocks to interact with the hypervisor, for example being able to compile a kernel module or the required C header files to use the definitions of structures that are passed as arguments to the hypercall. We use the Xen Testing Framework [32], which provides these building blocks, supports many hypercalls and allows the user to create a security test easily. XTF also already has tests for past XSAs. For the XSAs we use, actual tests can be found via the following URL: https://xenbits.xen.org/gitweb/?p=xtf.git;a=blob_plain;f=tests/xsa-200/main.c.[3]

1) Extract Guest Interaction From Traces: To extract the guest interaction, we have to follow the variables that are used as parameters for hypercalls and the `copy_from_guest` function.[4] Given our abstraction, CBMC is allowed to choose arbitrary values for the parameters and returns from `copy_from_guest`, which matches that adversarial model: an adversarial guest can write arbitrary data to those.

We now use an example trace that has been generated for XSA 227, with the relevant parts shown in Figure 7, to turn it into the basic building blocks for a security test. Note, CBMC traces

[3]Replace the 200 with the any of the XSA numbers 200, 212, 213 or 227.
[4]In Xen 4.8, `copy_from_guest` is a macro, which is expanded to `copy_from_user_hvm`.

first list the function call, and then show how the parameters are evaluated.[5]

We implemented a harness that calls the hypercall `do_grant_table_op` (line 4), and which allows CBMC to choose arguments for this hypercall. CBMC picks cmd=0 (step 3), which is equivalent of the operation `GNTTABOP_map_grant_ref`.

CBMC also generates a pointer value for the variable uop (step 3), which is likely to be wrong. A wrong value is chosen, because the trace starts in the middle of a running Xen, and the initialization of the data structures for the guest in the hypervisor has not been done yet. To make the security test complete, a grant table map would have to be created first, and then the correct value would have been known. This work does not take the preparation of the environment into account yet, as that step requires further expert knowledge. In the same spurious way, the value for variable count is set to 258, but the trace actually consumes only a single element. Therefore, this value should be set to 1 in the actual code.

Next, in step 6 we call the function `gnttab_map_grant_ref`, following the value of the parameter cmd. This function then issues the call to `copy_from_guest` (step 7), and looks for data of the type "struct gnttab_map_grant_ref". The result of this operation should contain the value 4089, or 0xFF9, for the member host_addr. The rest of the trace then shows how this value is used and propagated forward until the assertion is violated in step 13. The assertion basically checks whether the lower bits of the address are set to 0, which fails, because the least significant bit is set in the representation of 0xFF9.

Now that the relevant information is extracted from the trace, the corresponding block in a security test for the fix of XSA 238 should contain the following lines:

```
struct gnttab_map_grant_ref map = {
    .host_addr = 0xFF9,
    .flags = 21,
    .ref = 0,
    .dom = 0,
};
hypercall_grant_table_op(
        GNTTABOP_map_grant_ref, &map, 1);
```

Again, due to the abstraction, there are values from the trace that might be invalid, because the aggressive slicing drops calls to functions that might have checked these properties. Furthermore, values to be used might depend on the system setup, so that the values CBMC reports might only be used as a guideline. Still, the trace highlights that it is possible to trigger the security issue from the guest, and furthermore provides candidate data that can be used to generate a security test for the issue.

C. Practical Relevance

EC2 launched in 2006. The Xen project reported its first XSA in March 2011, and has since announced more than 300 XSAs—about three per month. AWS' Xen security team has provided feedback to the Xen security team on several occasions and reported four follow-up XSAs. Using this experience, we analyzed the five XSAs above as examples. The generated traces for four out of those proved to be good enough as building blocks for tests: the relevant hypercall, as well as relevant input values for the hypercall are present in the trace. When there are multiple traces, we use the trace with most precision, i.e. highest unwinding, distance, and direct paths included. Ultimately this runnable test is used to avoid future security regressions. Only the trace for XSA 213 requires further work, as CBMC only reports the second half of the trace, and skips setting the hypervisor into a specific mode first, because we start the analysis from a non-deterministic machine state.

Compared to the manual assessments that we performed, the automated approach is often faster. If we take setting up Xen for the analysis into account, i.e. adapting available harnesses and packaging for AWS Batch, we are usually in the same ball-park. While AWS security experts can often quickly assess reachability of a location in the code base, for a specific configuration and version of the hypervisor, finding input values to bypass checks and doing this analysis for all different production configurations is more challenging. For these cases, the presented approach is a win, as the amount of engineering time spent can be reduced from engineering days to hours.

We note that failing to find a trace is not sufficient to allow the security team to ignore an issue, and the security team makes risk-based assessments based on their experience and judgment. In some cases in this scenario it is necessary to fall back to the traditional techniques to establish high confidence that a potential issue does not require a fix. In these scenarios, our tool can, nevertheless, still be a useful datapoint that, combined with other information, can give the security team confidence that additional investigation is not urgent.

The recent XSA 296 reported the vulnerability for PV and HVM guests. After about a day of manual work, AWS experts could rule out HVM guests. The new approach would have been helpful, as all reported traces would have relied on PV guest features, helping to rule out HVM faster.

For similarly complex software projects, the presented technique will provide valuable insight into reachability and produce sample input values to trigger a software bug. Less experienced teams might benefit from the automated approach to speed up response time for security incidents.

1) Limitations and open challenges: Our approach has a number of limitations which reduce the precision of our results. We may miss traces for several reasons: CBMC has an incomplete understanding of assembly code; we do not consider whether functions we remove modify globals; we unwind loops to a finite bound; and CBMC does not maintain a full model of memory. We may produce spurious traces because we start from a mid-point in the code using a non-deterministic start state. Addressing these limitations and running bit-precise analysis on large code bases remains an open challenge. We feel that the natural next step towards this would be refinement of our approximations.

[5]This section is based on the trace file xsa227.22701.trace; irrelevant lines are omitted.

```
----------------------------------------------------------------------
Function call: my_granttable_init (depth 1)
----------------------------------------------------------------------
Function call: textbf{do_grant_table_op} (depth 2)
----------------------------------------------------------------------
State 943: textbf{cmd=0u} (0x0)
State 944: uop={ .p=INVALID-65535 } ({ 0xFFFF8200 80000002 })
State 945: count=258u (0x102)
----------------------------------------------------------------------
Function call: gnttab_map_grant_ref (depth 3)
----------------------------------------------------------------------
State 999: copy_from_user_hvm(_d, (_s + i),
(sizeof(textbf{struct gnttab_map_grant_ref}) * 1));
 op={ textbf{.host_addr=4089ul}, .flags=21u, .ref=0u, .dom=0,
               .status=0, .handle=0u, .dev_bus_addr=0ul }
           ({ 0xFF9, 0x15, 0x0, 0x0, 0x0, 0x0, 0x0 })
----------------------------------------------------------------------
Function call: __gnttab_map_grant_ref (depth 4)
Function call: create_grant_host_mapping (depth 5)
----------------------------------------------------------------------
State 1144: {addr=4089ul} (0xFF9)
----------------------------------------------------------------------
Function call: create_grant_pte_mapping (depth 6)
----------------------------------------------------------------------
State 1194: {pte_addr=4089ul} (0xFF9)
----------------------------------------------------------------------
Violated property: assert((pte_addr & sizeof(l1_pgentry_t)-1)==0)
----------------------------------------------------------------------
```

Figure 7. The relevant parts of the CBMC trace for XSA 227 that guides test generation.

There are cases where security issues can only be triggered if there are two guest CPUs available. Analysis would have to scale for parallel interaction, including parallel hypercalls, and modification of guest data.

Finally, automation would benefit from an incremental approach of the technique. The investigating engineer might receive the initial results, and then modify the harness to restrict the search space. Today, the complete process has to be triggered again, and the whole search has to be repeated. We expect starting analysis from a state similar to a given trace to significantly reduce the run time of subsequent iterations, similarly to incremental SAT solving [33].

VI. Conclusion

We have described the application of bounded model checking and slicing to the Xen hypervisor used in triaging reported security concerns. By introducing improved handling of function pointers and approximating program slicing in combination with havocking functions we are able to use bounded model checking to construct counterexample traces that reproduce security issues and ultimately, determine their severity. Despite the open challenges listed in the previous section, we have shown that we can use automation to help generate security tests from patches today, which supports more rapid security analysis and also leads to a more secure cloud environment for customers. This tooling assists experts in determining the severity of security vulnerabilities, constructing security tests for these scenarios, and helping on-call security experts quickly decide whether they should wake up developers or allow them to enjoy their well-deserved sleep.

Acknowledgments: E. Polgreen is supported by NSF grants 739816 and 183713.

References

[1] A. Follner, A. Bartel, H. Peng, Y. Chang, K. K. Ispoglou, M. Payer, and E. Bodden, "PSHAPE: automatically combining gadgets for arbitrary method execution," in *Security and Trust Management - 12th International Workshop, STM 2016, Heraklion, Crete, Greece, September 26-27, 2016, Proceedings*, ser. Lecture Notes in Computer Science, vol. 9871. Springer, 2016, pp. 212–228. [Online]. Available: https://doi.org/10.1007/978-3-319-46598-2_15

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *SOSP*. ACM, 2003, pp. 164–177.

[3] Xenproject.org Security Team, "Xen security advisories," 2019, accessed: 2019-12-17. [Online]. Available: https://xenbits.xenproject.org/xsa/

[4] E. Polgreen, "CBMC extensions," 2018, accessed: 2019-12-19. [Online]. Available: https://github.com/polgreen/cbmc/tree/xen_extended_cbmc

[5] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 2988. Springer, 2004, pp. 168–176.

[6] D. Kroening and M. Tautschnig, "CBMC – C bounded model checker (competition contribution)," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 8413. Springer, 2014, pp. 389–391.

[7] J. Gallagher, R. Gonzalez, and M. E. Locasto, "Verifying security patches," in *Workshop on Privacy & Security in Programming*. ACM, 2014, pp. 11–18.

[8] R. Wojtczuk, "UQBTng: A tool capable of automatically finding integer overflows in Win32 binaries," in *CCC*. Chaos Communication Congress, 2005.

[9] A. Vasudevan, S. Chaki, L. Jia, J. M. McCune, J. Newsome, and A. Datta, "Design, implementation and verification of an extensible and modular hypervisor framework," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2013, pp. 430–444.

[10] J. Franklin, S. Chaki, A. Datta, J. M. McCune, and A. Vasudevan, "Parametric verification of address space separation," in *POST*, ser. Lecture Notes in Computer Science, vol. 7215. Springer, 2012, pp. 51–68.

[11] E. Alkassar, M. A. Hillebrand, W. J. Paul, and E. Petrova, "Automated verification of a small hypervisor," in *VSTTE*, ser. Lecture Notes in Computer Science, vol. 6217. Springer, 2010, pp. 40–54.

[12] W. J. Paul, S. Schmaltz, and A. Shadrin, "Completing the automated verification of a small hypervisor – assembler code verification," in *SEFM*, ser. Lecture Notes in Computer Science, vol. 7504. Springer, 2012, pp. 188–202.

[13] M. Dahlin, R. Johnson, R. B. Krug, M. McCoyd, and W. D. Young, "Toward the verification of a simple hypervisor," in *ACL2*, ser. EPTCS, vol. 70, 2011, pp. 28–45.

[14] A. Puccetti, "Static analysis of the XEN kernel using Frama-C," *J. UCS*, vol. 16, no. 4, pp. 543–553, 2010.

[15] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*. USENIX Association, 2008, pp. 209–224.

[16] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO '04. USA: IEEE Computer Society, 2004, p. 75.

[17] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Commun. ACM*, vol. 57, no. 2, pp. 74–84, 2014. [Online]. Available: https://doi.org/10.1145/2560217.2560219

[18] B. Chen, Z. Yang, L. Lei, K. Cong, and F. Xie, "Automated bug detection and replay for COTS linux kernel modules with concolic execution," in *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*, K. Kontogiannis, F. Khomh, A. Chatzigeorgiou, M. Fokaefs, and M. Zhou, Eds. IEEE, 2020, pp. 172–183. [Online]. Available: https://doi.org/10.1109/SANER48275.2020.9054797

[19] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 1579. Springer, 1999, pp. 193–207.

[20] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B. Yassour, "The turtles project: Design and implementation of nested virtualization," in *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*. USENIX Association, 2010, pp. 423–436. [Online]. Available: http://www.usenix.org/events/osdi10/tech/full_papers/Ben-Yehuda.pdf

[21] H. Katzan Jr., "Operating systems architecture," in *AFIPS*, ser. AFIPS Conference Proceedings, vol. 36. AFIPS Press, 1970, pp. 109–118.

[22] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[23] A. Kivity, "KVM: Kernel-based virtual machine," 2006, accessed: 2019-12-19. [Online]. Available: http://lkml.iu.edu/hypermail/linux/kernel/0610.2/1369.html

[24] Xenproject.org Security Team, "Advisory XSA-227," 2018, accessed: 2019-12-19. [Online]. Available: https://xenbits.xenproject.org/xsa/advisory-227.html

[25] D. Beyer, "Software verification with validation of results - (report on SV-COMP 2017)," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 10206, 2017, pp. 331–349.

[26] M. Weiser, "Program slicing," in *ICSE*. IEEE Computer Society, 1981, pp. 439–449.

[27] A. Biere, E. M. Clarke, R. Raimi, and Y. Zhu, "Verifiying safety properties of a power PC microprocessor using symbolic model checking without bdds," in *CAV*, ser. Lecture Notes in Computer Science, vol. 1633. Springer, 1999, pp. 60–71.

[28] M. Shapiro and S. Horwitz, "Fast and accurate flow-insensitive points-to analysis," in *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*. ACM Press, 1997, pp. 1–14. [Online]. Available: https://doi.org/10.1145/263699.263703

[29] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT*, ser. Lecture Notes in Computer Science, vol. 2919. Springer, 2003, pp. 502–518.

[30] Xen Project, "Xen project 4.8.0," 2018, accessed: 2019-12-19. [Online]. Available: https://xenproject.org/downloads/xen-project-archives/xen-project-4-8-series/xen-project-4-8-0/

[31] "AWS Batch," 2016, accessed: 2019-12-19. [Online]. Available: https://aws.amazon.com/batch

[32] "Xen test framework," http://xenbits.xen.org/docs/xtf/, accessed: 2018-09-10.

[33] N. Eén and N. Sörensson, "Temporal induction by incremental SAT solving," *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 4, pp. 543–560, 2003.