Secure Virtualization with Formal Methods

Cynthia Sturton

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-224
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-224.html

December 18, 2013

Secure Virtualization with Formal Methods

by

Cynthia Koren Levine Sturton

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor David Wagner, Chair
Associate Professor Sanjit A. Seshia
Assistant Professor Brian Carver

Fall 2013

Secure Virtualization with Formal Methods

Abstract

Secure Virtualization with Formal Methods

by

Cynthia Koren Levine Sturton

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David Wagner, Chair

Virtualization software is increasingly a part of the infrastructure behind our online activities. Companies and institutions that produce online content are taking advantage of the "infrastructure as a service" cloud computing model to obtain cheap and reliable computing power. Cloud providers are able to provide this service by letting multiple client operating systems share a single physical machine, and they use virtualization technology to do that. The virtualization layer also provides isolation between guests, protecting each from unwanted access by the co-tenants. Beyond cloud computing, virtualization software has a variety of security-critical applications, including intrusion detection systems, malware analysis, and providing a secure execution environment in end-users' personal machines.

In this work, we investigate the verification of isolation properties for virtualization software. Large data structures, such as page tables and caches, are often used to keep track of emulated state and are central to providing correct isolation. We identify these large data structures as one of the biggest challenges in applying traditional formal methods to the verification of isolation properties in virtualization software.

We present a new semi-automatic procedure, S^2W, to tackle this challenge. Our approach uses a combination of abstraction and bounded model checking and allows for the verification of safety properties of large or unbounded arrays. The key new ideas are a set of heuristics for creating an abstract model and computing a bound on the reachability diameter of its state space. We evaluate this methodology using six case studies, including verification of the address translation logic in the Bochs x86 emulator, and verification of security properties of several hypervisor models. In all of our case studies, we show that our heuristics are effective: we are able to prove the safety property of interest in a reasonable amount of time (the longest verification takes 70 minutes to complete), and our abstraction-based model checking returns no spurious counter-examples.

One weakness of using model checking is that the verification result is only as good as the model; if the model does not accurately represent the system under consideration, properties proven true of the model may or may not be true of the system. We present a theoretical framework for describing how to validate a model against the corresponding source code, and an implementation of the framework using symbolic execution and satisfiability modulo theories (SMT) solving. We evaluate our procedure on a number of case studies, including the Bochs address translation logic, a component of the Berkeley Packet Filter, the TCAS suite, the FTP server from GNU Inetutils, and a component of the XMHF hypervisor. Our results show that even for small, well understood code bases, a hand-written model is likely to have errors. For example, in the model for the Bochs address translation logic – a small model of only 300 lines of code that was vigorously used and tested as part of our work on S^2W– our model validation engine found seven errors, none of which affected the results of the earlier effort.

To my husband, with love and gratitude.

Contents

List of Figures

List of Tables

Acknowledgments

I thank my advisor, David Wagner, for his support and guidance throughout my graduate studies. His keen insights and deep understanding of all things security strengthened my research, and I learned a tremendous amount from working with him. I am also grateful to him for helping me to develop collaborations with folks outside of Berkeley's computer science division. They have been integral to my growth and development as a researcher. Any success I have had is due in large part to those collaborations.

I thank my committee members, Brian Carver and Sanjit Seshia, and my collaborators on the work appearing in this thesis: Rohit Sinha, Michael McCoyd, Sakshi Jain, Thurston Dang, and Petros Maniatis. I would especially like to thank Sanjit. Through collaborations with him on this and other work, I have learned to appreciate the power, and limitations, of using formal verification in a security context. His encouragement and advice throughout the years has made him a valuable mentor and greatly enriched my graduate school experience.

Although our work together is not a part of this thesis, I would like to thank Sam King and Matthew Hicks for fun, and fruitful, collaborations over the years. I am particularly grateful to Sam for his interest in, and encouragement of my studies and future career.

I am grateful to Colleen Lewis for her friendship. We have spent hours together laughing, stressing, and working, and it has all made for a wonderful graduate school experience.

I am grateful to my family for their ongoing love and encouragement. And I am deeply grateful to my husband. He has supported me with love, kindness, generosity, and delicious food throughout the long and circuitous path that led to this point.

Chapter 1

Introduction

Virtualization software, such as CPU emulators, virtual machine monitors (VMM), or hypervisors (HV), provides many practical benefits. It typically sits below the operating system and adds a layer of indirection between the operating system and the hardware platform. This is useful for a variety of applications. It can be used to present an instruction set architecture different than that of the actual hardware platform, which is useful for the development of operating systems and applications for new platforms. It can be used to multiplex hardware resources to allow multiple operating systems to co-exist on one platform. And it provides a vantage point below the operating system for more complete monitoring and analysis of the system, which is useful in the development and testing of new operating systems, and for malware analysis.

Virtualization is useful; however, virtualization software is typically complex and executes with high privilege levels, making it especially vulnerable to attack. In this work, we seek to verify the correctness of security-critical components of virtualization software. We investigate the use of formal verification techniques to prove properties about the security of system virtualization software in order to increase the overall security of the systems that rely on it.

Virtualization software is well known for its role in cloud computing, and in particular for infrastructure-as-a-service (IaaS) style cloud computing. With IaaS, the cloud provider maintains a large data center with many servers and rents out compute time to its customers. A customer can execute their entire software stack for as long as necessary on one of the provider's servers, and they pay only for the compute time they use. IaaS services allow customers to grow and shrink their infrastructure on demand. For example, an online shopping site can increase their capacity to handle high demand during peak shopping times without the overhead of purchasing, configuring, and maintaining new hardware. During off-peak hours, the customer can easily reduce their capacity and their costs. Customers of IaaS include well known online shopping sites, news media organizations, universities, and social media sites. IaaS style cloud computing is the fastest growing segment of the cloud

computing market and is expected to reach $9 billion in 2013, up from $6.1 billion in 2012 [1].

Cloud providers are able to take advantage of economies of scale to provide cheap compute capacity by using virtualization software. Hypervisors and virtual machine monitors allow multiple operating systems to execute on a single hardware platform, and give each guest operating system the illusion that it is running alone on the hardware. Since the different guest operating systems may belong to different customers, possibly even adversarial customers, customers and cloud providers rely on virtualization software to maintain strict isolation between guest OSes. In addition to providing a multiplexed hardware platform, virtualization software enables other tools that make data centers efficient and practical for cloud providers. For example, live migration of running operating systems allow for efficient resource allocation [2], and verifiable accounting of resource use allows for efficient billing [3].

Beyond cloud computing, researchers have proposed using system virtualization software as a platform to increase the security of end-user machines [4, 5]. The isolation properties provided by hypervisors and virtual machine monitors can be used to implement red–green configurations on client desktops [6–9]. A red-green system allows the user to maintain a separation between their trusted and untrusted activities. One guest OS, the "green" one, is locked-down, trusted, has access to private data and a secure network, but has little access to the public Internet. A second, "red" guest OS has full access to the Internet, but not the secure network or private data.

Because the virtualization software sits beneath the OS layer in the system stack, it is well situated for OS introspection: it has a complete view of OS activity, and it protects itself from access by any code executing within the guest. A wide variety of malware detection and analysis systems based on virtual machine monitors, hypervisors, and emulators have been developed to take advantage of this property. Intrusion Detection Systems (IDS) usually have to choose between existing as a kernel module in the end system or sitting in the network, entirely outside the system. In the former case, the IDS has a complete view of the system, but it exists inside the system and so is vulnerable to attack from the very malware it aims to detect, rendering the IDS ineffective. In the latter case the IDS is inaccessible to the end system and therefore is protected from compromise, but it has an incomplete view of an end system's activity. With virtualization software, the IDS can exist in the virtualization layer below the OS. In this layer it has a complete view of the system it aims to protect, but is kept isolated and unreachable by the potentially compromised OS [10, 11]. This property makes virtualization software especially powerful for the detection of rootkits [12–14], which traditionally have controlled the lowest level of software, making them difficult to detect from within the infected OS.

Its powerful vantage point also makes virtualization software useful for malware analysis [15–23]. Malware can be allowed to run in the guest OS, while its behavior is inspected by the virtualization software. The virtualization software protects itself from any activity in the guest OS, preventing the malware from shutting down the analysis. Similar techniques have also been shown to be useful for auditing and debugging operating systems [24, 25]. In

addition to detection and analysis, virtualization software has also been used to prevent compromise of the guest OS by preventing any unauthorized code from executing while the guest is in kernel mode [26–32].

Rather than focus on protecting the guest operating system from infection, a number of researchers have taken the view that it is better not to trust the OS at all, and tools based on hypervisors, emulators and virtual machine monitors have been developed to protect user-level code and data from a malicious OS [33–36]. There has even been work done on using nested virtualization to increase the security of the hypervisor or virtual machine monitor itself [37, 38].

In all of these systems, the strength of the tool depends on the strong isolation and containment properties provided by the virtualization software. Because it sits below the operating system, and because it tends to have a considerably smaller code base than a commodity operating system, some researchers have suggested that virtualization software makes an ideal platform for building secure systems [4]. However, virtualization software is not necessarily immune from vulnerabilities. Although smaller than an operating system, the implementation of virtualization software can still be quite large. The popular Xen hypervisor is roughly 150 KLOC [39], and the KVM kernel module is roughly 42 KLOC.[1] The code is complex and previous research has found errors in popular virtualization software [40, 41]. Vulnerabilities that allow a guest to access the host's memory space have also been discovered [42–45].

The goal of this work is to prove the correctness of the isolation and containment properties that are fundamental to the security of so many tools and systems based on virtualization software. To do this, we focus on virtualization software's management of memory resources. The memory management components are responsible for providing to the guest the correct virtual-to-physical memory translation, which is usually provided by hardware. They must also provide the correct mapping from a guest operating system's view of physical memory to the machine's physical memory. The memory management code typically includes sets of page tables and caches. Proving safety properties about these large data structures presents a challenge to formal verification techniques. In this work, we focus on the verification of existing systems, rather than the development of a new hypervisor or emulator, and we use model checking to perform the verification.

In the first half of this thesis (Chapter 3), we consider the verification of safety properties in systems with large arrays and data structures, such as address-translation tables and other caches. These large data structures make automated verification based on straightforward state-space exploration infeasible. We present S^2W, a new abstraction-based model-checking methodology to facilitate automated verification of such systems. As a first step, inductive invariant checking is performed. If that fails, we compute an abstraction of the original system by precisely modeling only a subset of state variables. This subset of the state constitutes a "small world" hypothesis, and is extracted from the property. Finally, we verify the safety property on the abstract model using bounded model checking. We ensure

[1] Generated using David A. Wheeler's 'SLOCCount' http://sourceforge.net/projects/sloccount/.

the verification is sound by first computing a bound on the reachability diameter of the abstract model. For this computation, we developed a set of heuristics that we term the "short world" approach. We present several case studies, including verification of the address translation logic in the Bochs x86 emulator, and verification of security properties of several hypervisor models. Through our case studies we demonstrate that with our approach, model checking can be successfully applied to large table-like data structures; this removes one key barrier to automated verification of system virtualization software. The material in this chapter is based on joint work with R. Sinha, P. Maniatis, S. A. Seshia, and D. Wagner [46].

One limitation of this approach is that the verification is done on a model and not on the code directly. As a consequence, the result of verification is only as valid as the model; if the model does not accurately capture the behavior of the system, a property proven true of the model may or may not be true of the actual system. Therefore, it is essential to validate the model against the source code from which it is constructed. In the second half of this thesis (Chapter 4), we present a framework for validating the manually-built model against the code. The framework consists of two components. The first, data-centric model validation, checks that, for data structures relevant to the property being verified, all operations that update these data structures are captured in the model. The second, operation-centric model validation, checks that each operation is correctly simulated by the model. Both components are based on a combination of symbolic execution and satisfiability modulo theories (SMT) solving. We demonstrate the application of our methods on several case studies, including the model of the address translation logic in the Bochs x86 emulator that we verify in Chapter 3, the Berkeley Packet Filter, a TCAS benchmark suite, the FTP server from GNU Inetutils, and a component of the XMHF hypervisor. This demonstrates that it is possible to validate the model against the code and gain increased confidence that modeling errors have not affected our ability to find bugs in the code – making formal verification of system virtualization software all the more compelling. The material presented in this chapter is based on work done jointly with R. Sinha, T. H. Y. Dang, S. Jain, M. McCoyd, T. W. Yang, P. Maniatis, S. A. Seshia, and D. Wagner [47].

In summary, the thesis of this work is:

> Abstraction-based model checking techniques enable verification of large, symmetric data structures in software. In addition, the results of model checking can be strengthened through model validation techniques based on symbolic execution and SMT solving. These two results combine to provide an automated or semi-automated program verification technique suitable for proving security-critical isolation properties in virtualization systems.

We expect our results may have applications beyond the security of system virtualization software. Our work on S^2W is applicable to any system with large, table-driven data structures and gives us a new tool for dealing with the state space explosion problem in that

setting. And, model validation is a fundamental issue for all users of formal verification; our techniques may be broadly applicable to many applications of formal methods.

Chapter 2

Background

2.1 Virtualization Software

Virtualization software is a thin layer that sits below an operating system. It presents a virtualized hardware interface to the operating system above, introducing a level of indirection for the hardware–software interface. This intermediate layer can be used for many purposes: It can multiplex hardware resources, allowing multiple operating systems to run on a single platform. It can present to the operating system an instruction set architecture (ISA) that is different from the actual hardware ISA, allowing an OS and software compiled for one platform to be run on a different platform. And, this intermediate layer can provide an isolated execution environment in which it mediates all accesses to physical system resources. CPU emulators, virtual machine monitors (VMM), and hypervisors (HV) are all examples of virtualization software, and in this work we use "virtualization software" to refer to any of these software systems.

Virtual Machine Monitors and Hypervisors

Popek and Goldberg first formalized the definition of a VMM in 1974 [48]. They define a virtual machine (VM) as "an efficient, isolated duplicate of the real machine," and a VMM as the software that provides the VM environment. The VM is not any particular piece of software, rather it is the operating environment within which operating systems and applications may run. For example, the VM will include a virtual processor. To any software running in the VM, this processor will behave like a physical CPU. In reality, the virtual processor is a combination of the VMM software and the physical CPU hardware. Most instructions executed on the virtual processor are likely executed directly on the underlying physical CPU, but some are emulated by the VMM. The VMM can take advantage of the trapping mechanism provided by the physical CPU in order to intervene and emulate some

instructions when needed. All of this is done in a way that is transparent to software executing in the VM; the software executes on the virtual processor without being aware that it is virtual, and actually comprises both hardware and software.

Popek and Goldberg prescribe three properties that a VMM must provide: equivalence, resource control, and efficiency. Software running in the VM must produce the same effect as it would if it ran directly on the hardware, and not in the virtualized environment. The VMM must have control over hardware resources: the VM should not be able to access resources that have not been allocated to it and the VMM should be able to regain control of any resource already allocated to a VM. Software running in the VM environment must run efficiently. More specifically, a majority of the instructions executed must run directly on hardware without the intervention of the VMM. This last requirement differentiates VMMs from CPU emulators.

VMMs can be classified into two types [49]. A Type I VMM runs directly on the hardware platform without an underlying operating system. It has the highest level of privilege on the machine. Type II VMMs run on top of the host operating system, rather than on the bare metal. In both cases, the guest operating systems run in a layer above the VMM. Originally, "hypervisor" was another name for a Type I VMM. However, the lines between Type I and Type II VMMs are easily blurred and today the terms "hypervisor" and "VMM" are often used interchangeably, regardless of type. Some examples of well-known VMMs include VMware [50], Xen [51], KVM [52], and VirtualBox [53].

CPU Emulators

A CPU emulator is software that allows code compiled for one hardware platform to run on a different platform. For example, an application compiled for the PowerPC processor could be run on an x86 processor with the help of a CPU emulator. Emulators are not limited to use by applications; an entire operating system compiled for one architecture may be run on a different architecture with the help of a CPU emulator.

At its heart, an emulator works by translating the instructions of the guest binary code from the target (emulated) architecture to that of the host architecture. One way to manage the translation is by interpreting each instruction as it comes up. In this case, all CPU state, flags and control registers are implemented as variables in the emulation software and each instruction is implemented as a software function. Bochs is an example of a CPU emulator that uses interpretation [54]. A second type of translation works by recompiling the target binary code to the host architecture; recompilation is usually done one basic block of code at a time. QEMU is an example of a CPU emulator that uses recompilation [55].

Virtual machine monitors, hypervisors, and CPU emulators all work slightly differently, but they have in common the need to manage the often complex virtualization of system resources. In this work, we focus on memory and how virtualization software manages the

mapping from a guest operating system's view of memory to the machine's physical memory. The correct management of physical memory is key to providing the isolation between guest operating systems, or between the guest operating system and the host's execution that the virtualization software is often trusted to do. Virtualizing memory typically involves managing both page tables for address translation and caches of previously translated addresses. For example, Bochs will allow the OS to set up page tables with up to four levels of indirection, and it uses a translation lookaside buffer (TLB) with 1024 entries, each 160 bits wide, to cache previously translated addresses. It is the verification of these large data structures that we focus on in this work.

2.2 Model Checking

Throughout this work we use model checking-based techniques for our program verification. Model checking is a mature area of research, and there exist many variants. However, all model checking techniques have in common an approach to verification based on an exploration of the program's state space. A benefit of this approach is that if a property is disproven, the model checking engine can usually provide a counter-example showing the state, or series of states, that led to the failure.

Model checking was first introduced in the early 1980's [56–59] as a method for program verification that could be mostly automated. The original model checking algorithm is a graph-theoretic approach to program verification. The core idea is to represent the program to be verified as a directed graph, with nodes representing program states and edges representing transitions between states. Using a temporal logic such as Linear Temporal Logic [60] or Computational Tree Logic [56], properties about the program can be stated as properties about a path through the graph, or as properties about a sub-tree of the graph rooted at a particular node. Efficient graph exploration algorithms can then be used to prove or disprove the property.

While these early model checking algorithms can efficiently explore the state graph of a system, problems still arise for systems with large state spaces. The state space of a program is roughly exponential in the number of state variables. In software systems especially, the size and number of state variables can be large in the presence of large data structures. This "state space explosion" tends to limit the usefulness of so-called explicit state model checking, and various techniques have been developed to overcome it. One optimization is symbolic model checking. Rather than reasoning about individual states and transitions, symbolic model checking operates over sets of states and transitions. The sets, represented by Boolean formulas, represent the state space of the program symbolically. One way to store the Boolean formulas is as a Binary Decision Diagram (BDD). A BDD is a directed, acyclic graph that can efficiently represent many states; it is essentially a finite state automaton that takes as input a system state, represented as a sequence of binary digits, and accepts only those states that are reachable in the system. This type of symbolic model checking

can handle programs with a couple hundred variables, and state spaces ranging from 10^{30} to 10^{90} [61], whereas an explicit state model checking algorithm is restricted to state spaces that are small enough to be fully enumerated.

While BDD-based symbolic model checking can handle vastly larger designs than explicit state model checking, it still often can not scale to the sizes necessary for model checking software. Another symbolic model checking approach, bounded model checking [62], can handle much larger designs. In bounded model checking, the state space and property to verify are expressed as satisfiability problems and given to a SAT solver to either prove or disprove the property. SAT-based bounded model checking can scale to handle large designs. However, the drawback is that the state space of the program is explored only up to a certain depth. The behavior of the system past that depth is unknown.

Another approach for managing the large state space of software systems is to introduce some abstraction into the model of the program. In an abstraction, multiple program states are elided into one, and information about the program is lost. In a *sound abstraction*, a property proven true of the abstract model is true of the original program. However, the reverse may not be true: in a sound abstraction a safety property may be disproven for the abstract model even though it is actually true of the original program. Abstractions can be introduced by omitting some details of the original program from the model. For example, state variables deemed irrelevant for the verification task at hand may be left out of the model, reducing the total number of program states. Abstractions may also be introduced for systems with multiple, symmetric variables, processes, or data structures. In these cases a single instance might be modeled precisely, while the state and transitions of all other instances are conservatively over-approximated. For example, a single entry in a large array may be modeled precisely, while all other entries are modeled abstractly. In Chapter 3, we present a new technique for model checking systems with large data structures using a form of symmetry-based abstraction.

Although originally developed for the verification of hardware designs and other finite state systems, model checking techniques have successfully been applied to verifying software systems [63, 64]. Software model checking tools typically comprise an input or modeling language, a language for formalizing the properties, and the model checking engine. A typical work flow for model checking is shown in Figure 2.1. Given a program $\mathcal{S}$, the first step is to build a model $\mathcal{M}$ of the program using the input language of the model checking engine. At the same time the property to be proven, Φ, must be formalized using the specification language of the model checking engine. In our work, we focus on verifying safety properties, properties that state an invariant of the system. In this case, the model checking engine checks that Φ holds at all system states, or equivalently, that no bad state, in which Φ does not hold, is ever reachable. The model checking engine can output one of three results. The first is that the property is proven: the engine explored the entire state graph of the model and found no bad state. The second possible outcome is that the property does not hold. In this case, the model checking engine was able to find a reachable state in which Φ

was false, and typically, the model checking tool will return a counter-example. The third possible outcome is that the model checking engine was not able to prove the property holds, but neither was it able to find a bad state and corresponding counter-example. This result essentially means the model checking engine either timed out or ran out of memory before finding a counter-example or successfully proving the property.

Model checking is an automated technique that allows us to prove properties about software systems relatively quickly and without requiring significant expertise in formal methods. However, it does have its drawbacks. One concern is that verification is done on a model of the program, not on the program itself. Typically, the model is built manually and it is always possible for human error to introduce discrepancies between the model and the program. Even when a model checking tool operates directly on source code, it will internally construct a model of the code, and the verification often relies on manual pruning of the code plus manually created environment models. If a model does not correctly represent its program, any subsequent verification results do not accurately reflect the correctness of the program. In Chapter 4 we present a framework for validating a model against its program to address this weakness. A similar issue arises during code maintenance. Any updates to a verified software system will require updating and reverifying the model. Keeping the code and model synchronized throughout the lifetime of the system represents a serious engineering challenge. We do not address this issue directly, but rather point it out as evidence of the broad need for model validation research in general. Another consequence of manually built models is that the modeling effort involved in the verification of large system software, such as a hypervisor or CPU emulator, can be monumental. Techniques to automatically derive the model from the program would be useful and would mitigate that concern. In our work, we manage this limitation by focusing our verification efforts on one component of the system that is critical to security: the memory management subsystem.

A second concern is that the formal property Φ may not be the right property. That is, it may not accurately reflect the high level system property that we wish to verify. This is a concern with any formal verification technique, including model checking. Typically, verification is done with some high-level English language property in mind. However, the verification tool, in this case, the model checking engine, requires the property be formalized. Correctly capturing the intended property in a mathematical formalism can be difficult. One way to protect against this type of error is to clearly define the high level properties of interest at the beginning of the verification effort and spend some time developing the corresponding mathematical statement. We do this in Chapter 3, where we discuss the high level security goals for this verification effort and spend some time developing the related formal properties.

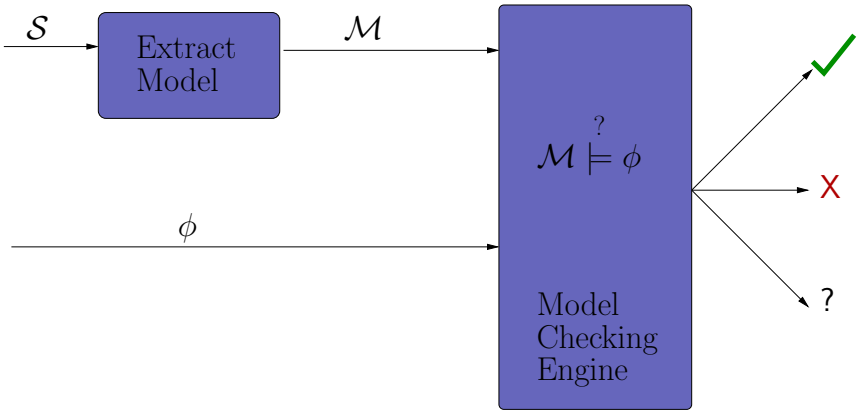

Figure 2.1: Model checking work flow.

Given a system $\mathcal{S}$, a model $\mathcal{M}$ is derived; $\mathcal{M}$, along with the property to verify, ϕ, are given to the model checking engine. The model checking engine can output one of three results: the property is proven to hold for the model, the property is proven not to hold, or the property can not be proven to hold, but neither has a counter-example been found before the engine times out.

2.3 Related Work

Formal Verification of Systems Software

Verifying the security of virtualization software has its roots in a long history of work verifying the security of operating systems. Dating back to the late 1970's, early verification efforts concentrated on proving properties related to process or data isolation [65]. Typically, the operating system was designed from the ground up, alongside the verification effort, and was based on a capability mechanism in order to make verification of isolation properties more tenable [66]. Formalizing the desired property was a large part of the research effort. The properties were then proven for an abstract, high-level specification of the system, and then as the specification was refined and details were added, each new, lower-level specification was formally shown to refine its predecessor. The last level of refinement was the implementation, either in a high-level language like C or in the executable machine language, and this implementation was proven to correctly implement the lowest-level specification. By transitivity, the property proven true of the abstract specification was shown to be true of the implementation. These early verification efforts were done using semi-automated, machine checked proofs. The Provably Secure Operating System (PSOS) [66], the UCLA Unix Security Kernel [67], and Kit [68,69] are all examples of these systems, known generally as "security kernels."

In 1981 Rushby introduced the concept of a separation kernel [70]. Imagined as a way to make the verification of secure operating systems easier, the basic idea of a separation kernel is to mimic logically the physical separation provided by a distributed system. In

a separation kernel, execution contexts are separated into partitions, data is kept isolated within its partition, and information flow between partitions is always mediated. Typically, two partitions can communicate only through a small set of communication channels, which are determined once at initialization and kept invariant thereafter. The behavior and data of one partition is kept entirely separate from, and can have no effect on, every other partition.

Separation kernels make verification of secure operating systems easier by separating the policy from its enforcement. A separation kernel is responsible for providing data isolation and mediation for all information flow between partitions. However, it is up to an initialization process (usually running above the kernel level) to determine how the partitions should be set up: how many partitions there should be, which processes should run in each partition, and which partitions should be allowed to communicate with each other. In other words, it is up to that initialization process to set up the policy that says what security properties the system should provide. In this way, a separation kernel can be verified to show it properly enforces the partitioning, while the security policy of a particular system can be independently verified to show the system's policy provides the desired security properties. Greve et al. formalized the separation policy that a general purpose separation kernel should provide [71].

More recently, separation kernels have been used in a variety of security- and safety-critical contexts. Baumann et al. demonstrate the verification of the memory manager in PikeOS, a separation kernel based on the L4 kernel [72]. They show that memory will never be misallocated between partitions, and threads can only access memory within their own partition. The verification is done using a combination of automatic and manual tools. They use the VCC verifier for concurrent C code to establish certain lemmas hold for the source code, then a manual proof combines the lemmas into a proof of the isolation properties.

Separation kernels have been successfully used as the basis for systems requiring Common Criteria certification of level EAL6 or EAL7, the highest levels possible. Richards demonstrated the formalization and verification of a separation kernel in a real-time operating system environment used in avionics [73]. The verified separation property ensures fault-containment: a fault occurring in one partition will have no effect on the behavior of any other partition. Martin et al. report on the formal specification and development of a separation kernel for use in a smartcard [74]. The kernel provides the basis for the key management system of an F22 fighter aircraft and for secure radios used by the navy. Heitmeyer et al. demonstrate the formalization of a data separation property, and its verification, for an embedded device [75, 76]. The kernel provides temporal as well as spatial data separation, as a partition's classification could change over time, and memory accessible at one classification level may not be accessible at a different classification level.

Perhaps the most complete verification, to date, of a modern, general purpose microkernel is the seL4 project [77, 78]. The authors provide complete functional verification down to the source code level of the kernel, and their verification of information flow security properties make seL4 applicable as a separation kernel. The kernel's functionality is expressive enough

to allow a complete paravirtualized Linux kernel to run in one partition. However, this high assurance does come at a cost. Partitions are determined statically, and any desired communication between partitions must also be set up statically at initialization. Scheduling of partitions is done in a pre-determined, set round-robin fashion, with each partition getting a fixed time slice. As such, although an excellent example of verification of system software, seL4 is not suitable for use as a general purpose hypervisor. For example, data centers use hypervisors to try to eke out the most efficient allocation of resources to guest operating systems with varying workloads.

In all these cases, the formalization of the separation property differs slightly to suit the operating context, but the main goal – verifying isolation between components – is similar to our efforts to verify isolation between guest operating systems in a virtualized environment. The above research differs from ours, though, in that all the above cases, the separation kernel and its verification were developed together, whereas we focus on the verification of legacy software systems.

Verification of Virtualization Software

In this section we discuss some of the previous research exploring the possibility of verifying virtualization software.

Barthe et al. built a model of memory management in a paravirtualized hypervisor based on a simplified version of Xen [79]. They use the Coq proof assistant [80] to verify three properties of the model: 1) Isolation: A guest operating system can only read or write its own memory. 2) Non-interference: the behavior of one guest operating system is not influenced by the behavior of any other guest. 3) Liveness: a request made by a guest operating system to the hypervisor will eventually get a response. The model and proofs took around 20 KLOC in Coq's input language.

The Xenon project takes a step toward the verification of a general purpose hypervisor [81–83]. Based on the Xen hypervisor, Xenon strips out some non-essential features to make verification easier. The authors developed a formal specification of a security policy that guarantees non-interference between domains. This specification served as a guiding document during the re-engineering effort. They also developed formal models of some parts of the hypervisor, such as the hypercall interface. Full verification of an information flow security policy in a commodity hypervisor not designed for verification is an ambitious goal, and development of the formal models and specifications are important first steps. However, the verification of the specified properties was not completed.

The Hyper-V/VCC project represents perhaps the most ambitious hypervisor verification project to date [84]. The Hyper-V hypervisor is a large general-purpose hypervisor comprising roughly 100 KLOC in C and 5 KLOC in assembly [85]. It was not designed for verification, rather the reverse: the verification tool, VCC, was specifically designed for use

on large software systems such as Hyper-V. VCC is a verifier for concurrent C code [86]. It requires code be annotated and relies on code contracts, such as function pre- and post-conditions to develop the verification formula that can then be sent to an SMT solver for proof of validity. Along the way, the authors of this project developed a "baby hypervisor," the verification of which they used to guide the development of VCC [87]. For both the baby hypervisor and Hyper-V, the authors concentrated their verification efforts on memory virtualization, in particular that the virtual TLB presented to a guest correctly emulates the behavior of a physical TLB [88].

Designing Virtualization Software for Verification

Formal verification of large software systems often runs into difficulty handling the complexity in these systems. The research described in this section tackles that problem by building systems specifically designed to make formal verification feasible. The designs focus on making the code modular, with well-defined interfaces, and often with functionality limited in some way that will considerably reduce complexity, thereby making verification easier.

Nova is an example of a hypervisor specifically designed for verification [89]. Similar to microkernels, much of the functionality of the hypervisor is moved to the user-level, minimizing the amount of privileged code that must be verified. Also similar to many microkernels, the Nova hypervisor is built around capability-based protection domains. The verification goals are to demonstrate spatial and temporal isolation between guest operating systems. The authors of Nova concentrate on tackling two challenges present in any verification effort: correct modeling of the system to be verified and correct representation of the system's environment. They tackle the first problem by developing a compiler from a subset of C++ to a formal semantics suitable for use as input to the PVS interactive theorem prover. This enables direct verification of the source code without requiring a modeling step. This is similar in some respects to the Hyper-V verification effort, which also included a translation from annotated C to a logical formula that could be then fed into a theorem prover. To model the environment, the Nova authors developed a formal description of parts of the IA32 architecture, including models of three memory interfaces provided by the physical CPU: physical RAM, memory-mapped devices, and virtual memory [90, 91]. Although verification of the Nova hypervisor was not completed, its design, along with the environment models provide a strong contribution to the field of hypervisor design-for-verification [92].

Another, recent effort of hypervisor design for verification is the XMHF hypervisor framework [93]. The framework is designed to be an easily extended platform for building security-critical, hypervisor-based applications. The authors make three design choices to enable verification of the framework and any hypervisors built on top of it: 1) Common hypervisor functionality is built into the XMHF core, so that it can be verified once and then used by any hypervisor built on the framework. 2) The framework relies on new hardware support for virtualization [94], including nested page tables [95,96], DMA protection [97,98], and support

for dynamic root of trust [99]. This obviates the need to verify complex software implementations of these systems. However, it does not remove the complexity; rather, it pushes the complexity down to the hardware level. 3) Hypervisors built on the framework are restricted to supporting a single guest. This allows the hypervisor to be sequential and single-threaded, and it allows the guest OS to directly control any peripheral hardware devices. Verification of the XMHF framework focuses on proving memory integrity, i.e., guaranteeing that no guest can modify any of the hypervisor's code or memory. This is done using the CBMC model checker; however, some aspects of the code base, such as the logic for looping over the large page table structures, can not be handled by CBMC and for those portions, manual auditing is used to give confidence in the code's correctness. Managing large data structures during verification is a well-known challenge, and in Chapter 3 we present one solution for how to formally verify such structures.

Testing Virtualization Software

Another way to validate the correctness of virtualization software is by testing. With testing, the focus is on bug finding, rather than proving correctness. And, although the state space of these systems is by far too large to allow testing to be complete, several methods have been developed which have proven effective. Ormandy first used black box fuzz testing to look for security vulnerabilities in four virtualization software systems: Bochs, QEMU, VMware, and Xen [100]. He focused on the instruction decoding and handling mechanisms and used randomly generated inputs to find instructions or I/O activity that would cause the emulator to crash or exit abnormally. He found bugs in all four systems tested, and the bugs ranged from buffer and heap overflow errors to divide-by-zero errors. Ormandy showed that, for each system tested, an attacker could exploit the vulnerabilities found to reliably halt the virtualization process and, in some cases, take over the host process to run arbitrary code on the host. The latter is clearly dangerous, and the former can be used by the attacker to thwart malware analysis.

Martignoni et al. used directed fuzzing combined with differential testing to find bugs in process emulators, system emulators, and virtual machines [41]. In all cases, testing focused on finding defects in the emulation software, i.e., test cases or instructions that resulted in the emulated state differing from what the true state would have been. One of the challenges in using fuzzing-based methods on emulation software is achieving high instruction coverage; a random sampling of byte-code patterns is unlikely to achieve full x86 instruction coverage. A second challenge is determining, for each test case generated, what the correct system state should be after executing the instruction. Martignoni et al. tackle the first problem using a combination of purely random byte-code patterns and random data fields with known opcode fields. The authors tackle the second problem using a physical CPU as the oracle that gives the correct post-test case state. The authors looked at the system emulators Bochs, QEMU, VMware, and VirtualBox and found defects in all four of the systems. Paleari et al. applied a similar technique to x86 disassemblers, and again found bugs in all systems tested [101].

Disassembly code is similar to the instruction decode algorithms found in CPU emulators and hypervisors.

A different approach to testing virtualization software was taken by Martignoni et al. with their technique dubbed "path lifting" [40]. The authors use symbolic execution to explore paths through one "high-fidelity" emulator and use the results to generate test cases for a second "low-fidelity" emulator, looking for places where the behavior between the two emulators diverges. This approach is more complete than fuzz-based testing, but is still primarily concerned with bug finding. Like all the testing based techniques discussed in this section, path lifting is effective at finding bugs, but is unable to prove the absence of bugs or prove the validity of safety properties. That is the essential trade-off made between testing and formal verification: testing allows more automation at the expense of less strong results.

Chapter 3

Verifying Large Data Structures using Small and Short Worlds

A particular challenge for the verification of CPU emulators, virtual machine monitors, and hypervisors is their use of large data structures. For example, logical-to-physical address translation requires data structures to store the CPU's Translation Look-aside Buffer (TLB) and page tables. While these structures are finite-length for any given processor, they are usually too large to represent precisely for verification; often, they are abstracted to be of unbounded length. The data structures in the resulting model of the system are thus *parametrized*: the indices into those structures are *parameters*, taking values in a very large or even infinite domain (typically finite-precision bit-vectors or the integers). The techniques proposed for verifying such parametrized systems fall into two classes: those based on a small-model or cut-off theorem (e.g., [102–104]), or those based on abstraction (e.g., [105–107]). While existing approaches are elegant and effective for their respective problem domains, they fall short for the problems we consider: the small-model approaches usually restrict expressiveness, while abstraction-based approaches either focus on control properties (as opposed to equivalence/refinement) or handle only certain kinds of data structures. In both cases, some of the realistic case studies we consider cannot be handled. (We make a fuller comparison in Section 3.5.)

In this chapter, we present a new semi-automatic methodology for verifying safety properties in systems with large data structures [46]. Our approach comprises three steps. First, we employ standard mathematical induction to verify the safety property, and if that succeeds, the process is complete. Second, if induction fails, we create an over-approximate abstraction of the system, the "small world," in which unbounded data structures are parametrized and, in general, only a subset of the state is updated as per the original transition relation (e.g., only a few entries of the unbounded data structures); the rest of the state is updated with arbitrary values at each step. With this abstraction, the model is more amenable to state-space exploration. Third, we attempt to find a bound k on the reachability diameter of

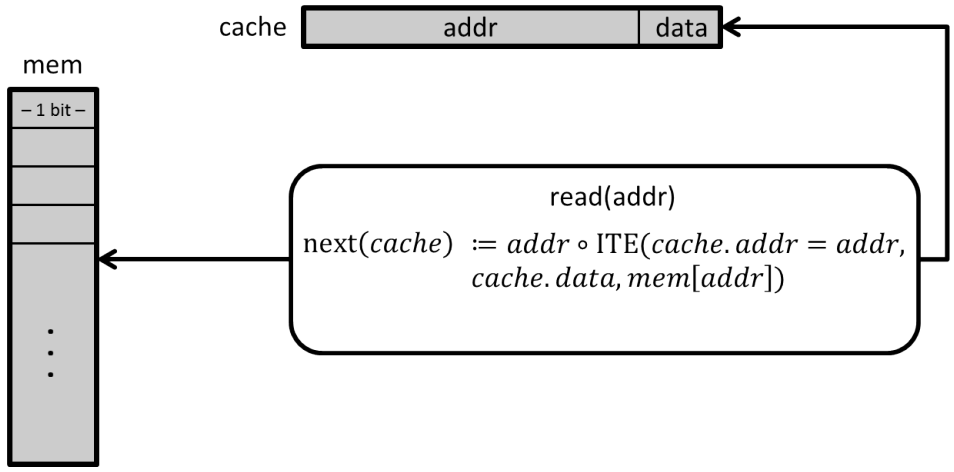

Figure 3.1: Running example.

A read-only memory and a single-entry cache. The cache is updated on each read command.

the small world so that, if bounded model checking (BMC) for k steps succeeds in the small world, then the safety property must hold in the small world, and since that is an over-approximation of the original system model, then the safety property holds there as well. Heuristics are presented for finding k that are effective for the class of systems we consider. We term this BMC-based approach the "short world" method, since it relies on computing a "short" bound for BMC. Our overall approach, termed *Small-Short-World (S^2W)*, is implemented on top of the UCLID system [108], which verifies abstract, term-level models using satisfiability modulo theories (SMT) solving. Note that the temporal safety verification problem for our class of systems is undecidable. As a result, S^2W is a semi-decision procedure.

3.1 Running Example

We introduce here a running example: a simple read-only memory system with a single-entry cache. We prove an invariant about the value returned by a read command. We build a model in our modeling language and demonstrate the verification of the safety property using S^2W. Our example is meant to be small, understandable, and illustrative, rather than "real-world."

Our example system (Figure 3.1) takes only one command, *read*, with a single parameter, the 32-bit address to be read; it returns a single-bit data value. At each read command, the cache is first checked. If the cache contains the data for the address requested, that value is returned. Otherwise, the value is read from memory. In either case, the cache is updated with the requested address and the returned data value. The update to cache is shown in the

above figure (we use "$\circ$" to mean concatenation). We prove an invariant about the cache: if the cache holds a valid address, then the cached data value is equal to the value stored in memory at that address. In other words, we show that the cache is correct.

3.2 Formal Description of the Problem

Notation and Terminology

A system is modeled as a tuple $\mathcal{S} = (\mathcal{I}, \mathcal{O}, \mathcal{V}, \mathit{Init}, \mathcal{A})$ where

- $\mathcal{I}$ is a finite set of input variables;
- $\mathcal{O}$ is a finite set of output variables;
- $\mathcal{V}$ is a finite set of state variables;
- Init is a set of initial states; and
- $\mathcal{A}$ is a finite set of assignments to variables in $\mathcal{V}$. Assignments define how state variables are updated, and thus define the transition relation of the system.

Input and output variables are assumed combinational (stateless), without loss of generality. $\mathcal{V}$ is the only set of state-holding variables. Variables can be of two types: *primitives*, such as Boolean or bit-vector; and *memories*, which includes arrays, content-addressable memories (CAMs), and tables. An output variable is a function of $\mathcal{V} \cup \mathcal{I}$. When representing a system without outputs, we will omit $\mathcal{O}$ from the representation. The set of initial states, Init, can either be viewed as a vector of symbolic terms representing any initial state, or as a Boolean-valued function of assignments to $\mathcal{V}$, written $\mathit{Init}(\mathcal{V})$.

Figure 3.2 denotes the grammar for expressions in our modeling language. The language has three expression types: Boolean, bit-vector, and memory.

$$
\begin{array}{ll}
b\mathrm{E} ::= & \texttt{true} \mid \texttt{false} \mid b \mid \neg b\mathrm{E} \mid b\mathrm{E}_1 \vee b\mathrm{E}_2 \\
& \mid b\mathrm{E}_1 \wedge b\mathrm{E}_2 \mid b\mathrm{v}\mathrm{E}_1 = b\mathrm{v}\mathrm{E}_2 \mid \mathrm{bvrel}(b\mathrm{v}\mathrm{E}_1, \ldots, b\mathrm{v}\mathrm{E}_k) \quad (k \geq 1) \\
& \mid \mathit{UP}(b\mathrm{v}\mathrm{E}_1, \ldots, b\mathrm{v}\mathrm{E}_k) \quad (k \geq 0) \\
b\mathrm{v}\mathrm{E} ::= & c \mid v \mid \mathit{ITE}(b\mathrm{E},\ b\mathrm{v}\mathrm{E}_1,\ b\mathrm{v}\mathrm{E}_2) \mid \mathrm{bvop}(b\mathrm{v}\mathrm{E}_1, \ldots, b\mathrm{v}\mathrm{E}_k)\ (k \geq 1) \\
& \mid m\mathrm{E}(b\mathrm{v}\mathrm{E}_1, \ldots, b\mathrm{v}\mathrm{E}_l) \mid \mathit{UF}(b\mathrm{v}\mathrm{E}_1, \ldots, b\mathrm{v}\mathrm{E}_k) \quad (l \geq 1, k \geq 0) \\
m\mathrm{E} ::= & A \mid M \mid \lambda(x_1, \ldots, x_k).b\mathrm{v}\mathrm{E} \quad (k \geq 0)
\end{array}
$$

Figure 3.2: Expression Syntax.

c and v denote a bit-vector constant and variable, respectively, and b is a Boolean variable. bvop denotes any arithmetic/bitwise operator mapping bit-vectors to bit-vectors, while bvrel is a relational operator other than equality mapping bit-vectors to a Boolean value. *UF* and *UP* denote an uninterpreted function and predicate symbol respectively. A and M denote constant and variable memories. $x_1, \ldots, x_k$ denote parameters (typically indices into memories) that appear in bvE.

The simplest Boolean expressions (bE) are the constants `true` and `false` or Boolean variables b; more complicated expressions can be constructed using standard Boolean operators or using relational operators with bit-vector expressions. We also allow a Boolean expression to be an application of an uninterpreted predicate to bit-vector expressions.

Bit-vector expressions (bvE) include bit-vector constants, variables, if-then-else expressions (ITE), and expressions constructed using standard bit-vector arithmetic and bitwise operations. Additionally, bit-vector expressions can be constructed as applications of uninterpreted functions returning bit-vector values and applications of memories to bit-vector arguments. Each bit-vector expression has an associated bitwidth. When a bit-vector expression is used as an index into a memory we may abstract the bitwidth to be unbounded, meaning that the memory is of arbitrary size.

Finally, the primitive memory expressions (mE) can be (symbolic) constants or variables. More complex memory expressions can be modeled using the Lambda notation introduced by Bryant et al. [108] for term-level modeling; this includes the standard `write` (`store`) primitive for modeling arrays, as well as more general operations such as parallel updates to arrays, operations on CAMs, queues, and other data structures.

A next-state assignment α denotes assignment to a state variable and is a rule of the form $\texttt{next}(x) := e$, $\texttt{next}(x) := \{e_1, e_2, \ldots, e_n\}$, or $\texttt{next}(x) := \{*\}$, where x is a signal in $\mathcal{V}$, and $e, e_1, e_2, \ldots, e_n$ are expressions that are a function of $\mathcal{V} \cup \mathcal{I}$. The curly braces express non-deterministic choice. The wildcard "$*$" is also an expression of non-deterministic choice. It is translated at each transition into a fresh symbolic constant of the appropriate type. The set of all next-state assignments defines the transition relation $\mathcal{R}$ of the system. Formally, $\mathcal{R} = \bigwedge_{\alpha \in \mathcal{A}} r(\alpha)$, where $r(\texttt{next}(x) := e) \doteq (x' = e)$ and $r(\texttt{next}(x) := \{e_1, e_2, \ldots, e_n\}) \doteq \bigvee_{i=1}^{n} (x' = e_i)$, where x' denotes the next-state version of variable x. We will sometimes write the transition relation as $\mathcal{R}(\mathcal{V}, \mathcal{I}, \mathcal{V}')$ to emphasize that it relates current-state variables $\mathcal{V}$ and next-state variables $\mathcal{V}'$ based on the inputs $\mathcal{I}$ received.

Example 1 We formally describe our model from Section 3.1. Let $\mathcal{S}_T = (\mathcal{I}, \mathcal{O}, \mathcal{V}, \mathit{Init}, \mathcal{A})$ be the system, with

- $\mathcal{I} = \{\mathit{addr}\}$. *addr* is the 32-bit address to read from memory.
- $\mathcal{O} = \{\mathit{out}\}$. *out* is the value read from either memory or the cache.
- $\mathcal{V} = \{\mathit{mem}, \mathit{cache}\}$. *mem* is constant and is modeled as an array of one-bit bit-vectors. It is represented by an uninterpreted function that maps a 32-bit address to a single bit. *cache* is a single 33-bit bit-vector; it holds the one-bit data value and 32-bit address of that value.
- $\mathit{Init} = (\mathit{mem}_0, \mathit{cache}_0)$. *mem* is initialized to hold arbitrary data values at each address. *cache* is initialized to hold an invalid address, 0x00000000, with an arbitrary data value.

- $\mathcal{A}$. On each read command *cache* is updated with the address read and the value returned by the read; *mem* remains constant.

Problem Definition

Consider a system $\mathcal{S}$ modeled as described in the preceding section. We similarly model the environment $\mathcal{E}$ that provides the inputs for $\mathcal{S}$ and consumes its outputs. The composition of $\mathcal{S}$ and $\mathcal{E}$, written $\mathcal{S}\|\mathcal{E}$, is the model under verification, $\mathcal{M}$. The form of the composition depends on the context; we use both synchronous and asynchronous compositions. We will represent the closed system $\mathcal{M}$ as a transition system $(\mathcal{V}_\mathcal{M}, \mathit{Init}_\mathcal{M}, \mathcal{R}_\mathcal{M})$, where the elements respectively denote state variables, initial states, and the transition relation. In all of our examples, the environment $\mathcal{E}$ is stateless, generating completely arbitrary inputs to $\mathcal{S}$ at each step; thus $\mathcal{V}_\mathcal{M} = \mathcal{V}$, $\mathit{Init}_\mathcal{M} = \mathit{Init}$ and $\mathcal{R}_\mathcal{M} = \mathcal{R}$.

This paper is concerned with verification of temporal safety properties of the form $\mathbf{G}\ \Phi$, where $\mathbf{G}$ is the temporal operator "always" and Φ is a state invariant of the form

$$\forall x_1, \ldots, x_k.\ \phi(x_1, \ldots, x_k) \tag{3.1}$$

where ϕ is a Boolean expression following the syntax of bE. The parameters $x_1, \ldots, x_k$ are bit-vector valued, but usually too large to exhaustively case split on; therefore, it is common practice to abstract their bitwidths to be unbounded, and perform verification for memories of arbitrary size.

Example 2 In our running example, we verify $\mathbf{G}\ \Phi_{3.2}$, where

$$\begin{aligned}\Phi_{3.2} \doteq\ \forall x.\ (\mathit{addr} = x) \rightarrow \\ ((\mathit{cache.addr} = \mathit{addr} \wedge \mathit{cache.addr} \neq 0) \rightarrow \\ \mathit{cache.data} = \mathit{mem}[\mathit{addr}])\end{aligned} \tag{3.2}$$

The problem tackled by this paper, temporal safety verification for systems with large data structures, is formally defined as follows.

Definition 1 (Large Data Safety Verification) *Given a model $\mathcal{M}$ formed as a composition of system $\mathcal{S}$ and its environment $\mathcal{E}$, and a temporal safety property $\mathbf{G}\ \Phi$, determine whether or not $\mathcal{M}$ entails $\mathbf{G}\ \Phi$.*

This problem is known to be undecidable in general since a two-counter machine can be encoded in our formalism using applications of uninterpreted functions [109]. Hence, we can only devise a semi-decision procedure for the problem. In the next section, we describe such a procedure that is based on abstraction.

3.3 Methodology

S^2W is based on a combination of abstraction and bounded model checking (BMC). We tackle state-space explosion by abstracting away all but a small subset of the space of the system. We call this mostly abstracted system our "small world." The abstracted portion of the system can be considered as being updated with an arbitrary value ("$*$") at each step of execution. All other parts of the system are modeled precisely. Thus, this abstraction is a form of localization abstraction [110], where the localization is to small, finite portions of large data structures.

We check the safety property on the small world using BMC. To make BMC sound, we first find and prove the length of the diameter D of our small world to use as the bound – i.e., D is an integer such that every state reachable in $D+1$ steps is also reachable in D or fewer steps. Proving that a conjectured diameter D is correct is undecidable in our formalism [111]. The key to our approach is a set of heuristics that are effective in our chosen application domain of emulators, virtual machine monitors, and hypervisors. For our examples, the diameter of the mostly abstracted system is typically small; we therefore term this the "short world."

If BMC runs for D steps and does not find a violation of the safety property in our small world, then the original model is safe. If BMC finds a counter-example, we cannot say whether the property holds for the original model: BMC can return a spurious counter-example. Choosing the small world well reduces the likelihood of finding spurious counter-examples.

To summarize, there are two crucial pieces to our approach: choosing the right small world and proving the length of the short world. We discuss both of these in more detail below.

As an optimization, we prefix the above approach with an attempt to prove the safety property using one-step induction (on the original, non-abstract model, $\mathcal{M}$). If that succeeds, there is no need to continue on to S^2W's abstraction. This step can be generalized to perform k-step induction as needed.

For the presentation in this section, it is convenient to represent the system under verification $\mathcal{S}$ as a transition system $(\mathcal{I}, \mathcal{V}, \mathcal{R}, \mathit{Init})$ where the elements of the tuple have the same meanings as in Section 3.2. The environment $\mathcal{E}$ sets the values of the input variables in $\mathcal{I}$ at each step; in all our case studies, the inputs from $\mathcal{E}$ are completely unconstrained. Verification (using induction or BMC) is performed on the composition of $\mathcal{S}$ and $\mathcal{E}$.

Induction

First, S^2W attempts to prove the safety property using simple one-step induction on the non-abstract model $\mathcal{M}$. We check the validity of the following two formulas, as per standard

practice:

$$Init_{\mathcal{M}}(\mathcal{V}_{\mathcal{M}}) \rightarrow \Phi(\mathcal{V}_{\mathcal{M}}) \tag{3.3}$$

$$\Phi(\mathcal{V}_{\mathcal{M}}) \wedge \mathcal{R}_{\mathcal{M}}(\mathcal{V}_{\mathcal{M}}, \mathcal{V}'_{\mathcal{M}}) \rightarrow \Phi(\mathcal{V}'_{\mathcal{M}}) \tag{3.4}$$

If both checks pass, the verification is complete. We report "Property valid" and exit. If check 3.3 fails, the property is invalid. We report "Property invalid in initial state" and exit. If check 3.4 fails, we continue with S^2W, to find the small world.

Small World

The objective of this step is to identify a small portion of system state that we should model precisely during BMC. Everything else will be allowed to take on arbitrary values at each step of execution.

It is important to note that the soundness of S^2W does *not* depend on the choice we make for the small world; we could randomly select some portion of the state to model precisely, abstract everything else away, and if our three steps complete and verify the property, the property would be true of the original, non-abstracted system. However, choosing the small world wisely ensures that the short world is indeed short, which allows BMC to complete in a reasonable amount of time. A well-chosen small world also reduces the number of spurious counter-examples returned by the BMC step.

We present here a heuristic for choosing the small world when dealing with systems involving large or unbounded data structures. In our case studies, the heuristic found a small world whose short world was reasonable in length and for which no spurious counter-examples were returned by the BMC.

To select those state variables to model precisely, S^2W starts with the property $\mathbf{G}\ \Phi$, where Φ is of the form $\forall x_1, x_2, \ldots, x_n.\ \phi(x_1, x_2, \ldots, x_n)$. If we prove Φ by instantiating the quantifier with a completely arbitrary, symbolic parameter vector $(a_1, a_2, \ldots, a_n)$, that suffices to prove the original property. Thus, starting with the symbolic vector $(a_1, a_2, \ldots, a_n)$, we compute a dependence set $\mathcal{U}$ for the instantiated property $\phi(a_1, a_2, \ldots, a_n)$. $\mathcal{U}$ is a set of expressions involving state variables and the parameters $a_1, \ldots, a_n$ such that fixing the values of the expressions in this set fixes the value of the instantiated property. For variable M modeling a memory, these expressions typically involve indexing into M at a finite number of (symbolic) addresses. For a Boolean or bit-vector variable, either the variable is in $\mathcal{U}$ or not.

Typically, this set of expressions is derived syntactically by traversing the expression graph of the formula ϕ represented in terms of state and input variables (after performing certain simplifications).

Example 3 In our running example, recall that the property is $\mathbf{G}\ \Phi_{3.2}$ where:

$$\begin{aligned}\Phi_{3.2} \doteq\ & \forall x.\ (addr = x) \rightarrow \\ & ((cache.addr = addr \wedge cache.addr \neq 0) \rightarrow \\ & \quad cache.data = mem[addr])\end{aligned}$$

$\Phi_{3.2}$ has the form $\forall x.\ \phi(x)$. Instantiating x with a, a fresh symbolic constant, we can drop the quantifier and get $\phi(a)$, for which, by propagating the equality $addr = a$, we see that its value is determined by the expressions $mem[a]$ and $cache$. Thus, we use $\mathcal{U} = \{mem[a], cache\}$ as our dependence set.

Once we have computed $\mathcal{U}$, using the above heuristic or some other method, we can define our small world. Recall that $\mathcal{S}$ is represented as a symbolic transition system $(\mathcal{I}, \mathcal{V}, \mathcal{R}, Init)$. Let $\hat{\mathcal{R}}$ be a transition relation that differs from $\mathcal{R}$ by setting all state variables not in $\mathcal{U}$ to a non-deterministic value and leaving all others unchanged. Abusing notation slightly to use $\mathcal{U}$ wherever we use $\mathcal{V}$, this means that $\hat{\mathcal{R}}(\mathcal{U}, \mathcal{I}, \mathcal{U}') = \mathcal{R}(\mathcal{U}, \mathcal{I}, \mathcal{U}')$, and $\hat{\mathcal{R}}(\mathcal{W}, \mathcal{I}, \mathcal{W}') = \texttt{true}$ for $\mathcal{W} = \mathcal{V} \setminus \mathcal{U}$. Similarly, $\hat{Init}(\mathcal{U}) = Init(\mathcal{U})$ and $\hat{Init}(\mathcal{W}) = \texttt{true}$.

Then the abstracted small world is $\hat{\mathcal{S}} = (\mathcal{I}, \mathcal{V}, \hat{\mathcal{R}}, \hat{Init})$. $\hat{\mathcal{S}}$ is an overapproximate, abstract version of $\mathcal{S}$ that precisely tracks only the state in $\mathcal{U}$, and allows all other variables to change arbitrarily at each step of execution. Thus, the composition of $\hat{\mathcal{S}}$ and $\mathcal{E}$ is an overapproximate model $\hat{\mathcal{M}}$. It is important to note that if $\mathcal{M}$ was infinite-state, $\hat{\mathcal{M}}$ continues to remain so.

The next step is proving a short world for $\hat{\mathcal{S}}$ and using BMC on $\hat{\mathcal{M}}$ to verify the property.

Short World

The objective of this phase is to determine a bound on the diameter D of the abstract model $\hat{\mathcal{M}}$. For this section, we will assume that $\mathcal{E}$ is stateless, as is the case for all of our case studies; the approach extends in a straightforward manner for the general case. Thus, the diameter of $\hat{\mathcal{M}}$ is the same as that of $\hat{\mathcal{S}}$.

Suppose we believe the diameter to be $D \leq k$. To verify this bound, we check the validity of the following logical formula:

$$\begin{aligned}&\forall \mathcal{V}_0, \mathcal{V}_1, \ldots, \mathcal{V}_{k+1}, \mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_{k+1}. \\ &\quad \left[Init(\mathcal{V}_0) \wedge \bigwedge_{i=0}^{k} \hat{\mathcal{R}}(\mathcal{V}_i, \mathcal{I}_{i+1}, \mathcal{V}_{i+1})\right] \rightarrow \\ &\quad \left[\exists \mathcal{V}'_0, \mathcal{V}'_1, \ldots, \mathcal{V}'_k, \mathcal{I}'_1, \mathcal{I}'_2, \ldots, \mathcal{I}'_k. Init(\mathcal{V}'_0) \wedge \bigwedge_{i=0}^{k-1} \hat{\mathcal{R}}(\mathcal{V}'_i, \mathcal{I}'_{i+1}, \mathcal{V}'_{i+1}) \wedge \bigvee_{i=0}^{k} \mathcal{V}_{k+1} = \mathcal{V}'_i\right] \qquad (3.5)\end{aligned}$$

Since $\hat{\mathcal{R}}$ modifies state expressions outside $\mathcal{U}$ arbitrarily on each step, we can replace $\mathcal{V}$ everywhere in the above formula with $\mathcal{U}$, and obtain the actual convergence criterion that must be checked.

Nevertheless, checking the convergence criterion is undecidable for the class of systems we are interested in, due to the presence of uninterpreted functions, memories, and parameters with unbounded bitwidth [111]. The quantified formula in (3.5) is also hard to solve in practice. Therefore, quantifier instantiation heuristics must be devised to perform the convergence check. In this section, we present two such heuristics that have worked well for the range of case studies considered in this paper.

The Sub-Sequence Heuristic

The first heuristic checks that for any state reachable in $k+1$ steps using $k+1$ symbolic inputs to $\hat{\mathcal{S}}$, one can also reach that state using *some sub-sequence of length* $\leq k$ of those $k+1$ symbolic inputs. We can express the sub-sequence heuristic as performing a particular instantiation of the existential quantifiers in criterion 3.5, and checking the validity of the following formula that results:

$$\begin{aligned}
&\forall \mathcal{U}_0, \mathcal{U}_1, \ldots, \mathcal{U}_{k+1}, \mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_{k+1}, \mathcal{U}'_0, \mathcal{U}'_1, \ldots, \mathcal{U}'_k. \\
&\left[\mathit{Init}(\mathcal{U}_0) \wedge (\mathcal{U}_0 = \mathcal{U}'_0) \wedge \bigwedge_{i=0}^{k} \hat{\mathcal{R}}(\mathcal{U}_i, \mathcal{I}_{i+1}, \mathcal{U}_{i+1}) \right] \rightarrow \\
&\bigvee_{(\mathcal{I}'_1, \ldots, \mathcal{I}'_k) \prec (\mathcal{I}_1, \ldots, \mathcal{I}_{k+1})} \left[\bigwedge_{i=0}^{k-1} \hat{\mathcal{R}}(\mathcal{U}'_i, \mathcal{I}'_{i+1}, \mathcal{U}'_{i+1}) \wedge \bigvee_{i=0}^{k} \mathcal{U}_{k+1} = \mathcal{U}'_i \right]
\end{aligned} \tag{3.6}$$

Here the symbol $\prec$ denotes that $(\mathcal{I}'_1, \ldots, \mathcal{I}'_k)$ is a sub-sequence of $(\mathcal{I}_1, \ldots, \mathcal{I}_{k+1})$.

The intuition for the sub-sequence heuristic is that in many systems with large arrays and tables, locations in those tables are updated destructively based on the current input, meaning that past updates do not matter. The address translation logic in the emulators we have studied has this nature. Thus, for such systems, it is possible to drop from the input sequence inputs that have no effect on the $k+1$-st step.

Observe the quantifier alternation in criterion (3.5) has been eliminated in the stronger criterion (3.6). Thus, we can simply perform a validity check of an SMT formula in the combination of theories required by our model. If the sub-sequence criterion (3.6) holds, then so does (3.5). However, it is possible that criterion (3.6) is too strong, even when a short diameter exists. This scenario necessitates an alternative semi-automatic approach, described next.

The Gadget Heuristic

The *gadget heuristic* is an approach to instantiating the existential quantified variables in criterion (3.5) that is particularly useful for systems in which some state in $\mathcal{U}$ depends on the past history of state updates in a non-trivial manner. A gadget is a small sequence of state transitions manually constructed to generate some subset of all reachable system state. A *universal gadget set* is a set of such sequences that, in concert, can generate any reachable system state.[1] The length k of the longest gadget in the universal gadget set is then an upper bound on the diameter of the system.

In terms of the formula expressed as criterion (3.5), a gadget is a particular guess for a set of initial states $\mathcal{V}'_0$ (expressed symbolically) and a sequence $\mathcal{I}'_1, \mathcal{I}'_2, \ldots, \mathcal{I}'_l$ (for $l \leq k$) of symbolic input expressions to use. For a finite number of gadgets, the inner existential quantifier in criterion (3.5) can be replaced as a disjunction over all the formulas obtained by substituting the gadget expressions for $(\mathcal{V}'_0, \mathcal{I}'_1, \mathcal{I}'_2, \ldots, \mathcal{I}'_l)$. If this instantiated formula is valid, then so is the original formula (3.5).

We defer further discussion about gadget construction to Section 3.3, where we discuss its use on our running example.

Performing BMC

Once we have proven k is an upper bound on the length of the diameter of $\hat{\mathcal{S}}$, we run BMC on $\hat{\mathcal{S}}$ for k steps. If $\phi(a_1, \ldots, a_n)$ holds at each step of the simulation, then it follows that $\hat{\mathcal{S}}$ satisfies $\mathbf{G}\ \Phi$. Because $\hat{\mathcal{S}}$ is an overapproximation of all states reachable by $\mathcal{S}$, it follows that $\mathcal{S}$ satisfies $\mathbf{G}\ \Phi$.

If BMC fails, we return a "short" counter-example. The counter-example will be no longer than k. If this is a valid counter-example, the property does not hold. If it is a spurious counter-example, we can return to step two of S^2W and expand our set $\mathcal{U}$ to include more state variables and inputs. Such a strategy would be an instance of counter-example-guided abstraction refinement.

Restricted State Spaces

In some systems, we are interested in proving a safety property over a restricted state space, where the restriction can be captured by a predicate over state variables. The restriction predicate is often specified as an antecedent in the temporal safety property. Examples of such a restriction can be found in Section 3.4. In such cases, we note that it is enough

[1] Our gadgets are inspired by "state-generation gadgets," used for automated testing of CPU emulators from arbitrary but reachable initial states [40], and by gadgets identified for return-oriented programming, used to produce a Turing-complete command set for malicious exploits [112].

to compute a bound on the reachability diameter — the short world bound — under that restriction. It also sufficient to perform model checking under the restriction.

Example

To illustrate the above approach, we apply it to our example. In step one we attempt to prove property $\mathbf{G}\ \Phi_{3.2}$ by induction. For this, we perform the following two checks:

$$Init(mem, cache) \rightarrow \Phi_{3.2} \tag{3.7}$$

$$\Phi_{3.2} \wedge \mathcal{R}_T(\mathcal{V}, \mathcal{V}') \rightarrow \Phi_{3.2} \tag{3.8}$$

Check (3.7) passes, since the cache is initially empty. However, the induction step (check (3.8)) does not pass. Starting from a state in which $\Phi_{3.2}$ holds, it is possible to transition to a state in which $\Phi_{3.2}$ is violated. To see why this is so, consider the following state for the cache and two particular entries of *mem*:

$$mem[i] := a,\ \ mem[j] := b,\ \ cache.addr := i,\ \ cache.data := z$$

where $z \neq a$, the last read was for address j, and the output was b. This state is not reachable in our model, but one-step induction does not take this into account. Note that $\Phi_{3.2}$ holds in this state: for every $x \neq j$, the antecedent $(addr = x)$ of the property is false and therefore the property is true; when $x = j$, the nested antecedent $(cache.addr = addr \wedge cache.addr \neq 0)$ is false and therefore the property is true. In this state, a $read(i)$ command will hit in the cache and the output will be z, making the property evaluate to false in the next state.

Since simple induction failed for our toy example, we move to the next step, identifying the small world $\hat{\mathcal{S}}_T$. As described in Section 3.3, we introduce a fresh symbolic constant a for x, removing the $\forall x$ quantifier from the property. We then select $\mathcal{U}$ syntactically from the property to be the set of expressions $\mathcal{U} = \{mem[a], cache\}$. In $\hat{\mathcal{S}}_T$ the variables in $\mathcal{U}$ are updated according to the original model ($\mathcal{S}_T$). All other state variables (all entries of *mem* other than $mem[a]$) are made to be fully abstract: they are allowed to update to non-deterministic values on every step. The same symbolic constant is used throughout the following short world checks.

The last step of our verification is to identify a short world and then run BMC on the abstract model for the length of the short world. We describe the gadget heuristic here; the sub-sequence heuristic would also work, although it finds a slightly looser bound on the length of the diameter. To build the gadgets we enumerate the possible end-state valuations for the system's state variables (*cache*, $mem[a]$) and for each, determine how to get there from a possible starting state. Notice that we only need to consider $mem[a]$ and not all of *mem*. This is because in our small world, $\hat{\mathcal{S}}_T$, all entries of *mem* other than $mem[a]$ receive new arbitrary values at the end of each step, so we know they can hold any possible value at every step of any trace. In theory there are 2^{34} end-states: one for each possible value of

cache.addr times the two possible values of *cache.data* and $mem[a]$ each. However, for our property, we do not really care about the precise valuation of *cache.addr*, rather, we care about whether $cache.addr = a$ and whether $cache.addr = 0$. So we can abstract away the details of *cache.addr* and consider the following 16 ending states:

$$\begin{aligned}&\{cache.addr = a, cache.addr \neq a\}\\ &\times \{cache.addr = 0, cache.addr \neq 0\}\\ &\times \{cache.data = 0, cache.data = 1\}\\ &\times \{mem[a] = 0, mem[a] = 1\}\end{aligned}$$

Not all of the above 16 states are reachable, and in the end four gadgets are enough to reach all reachable states. Each gadget uses either one or two read commands. We build the gadgets with the appropriate values for *addr* and show they form a universal gadget set and therefore, that the short world has length two. We then perform BMC and verify that the property holds.

3.4 Evaluation

We have evaluated S^2W on six case studies and describe them here: the TLB of the Bochs x86 emulator, a set-associative cache, shadow paging in a hypervisor, hypervisor integrity for SecVisor [103], the Chinese Wall access-control policy in sHype [103], and separation in ShadowVisor [102]. We describe the first three in detail; the last three were verified using one-step induction and we describe them only briefly. The code for all of our models, along with their verification, is available online.[2] All experiments were performed using UCLID [113] with the Plingeling SAT solver [114] backend on a machine with 8 Intel Xeon cores and 4 GB RAM.

Bochs' TLB

Bochs [54] is an open source x86 emulator written in C++ that can emulate a CPU, BIOS, and I/O peripherals. It can be used to host virtual machines, sandbox operating systems, and emulate the execution of system-level or user-level x86 code. The code base is large and previous research has shown that manual analysis and testing, while useful, are not enough to guarantee the system's correctness [40, 41].

Bochs emulates virtual memory using paging, which includes logic to translate a virtual address (VPN) to a physical address (PPN). Figure 3.3 illustrates the steps of a page walk. The input virtual address *vaddr* is partitioned into 3 sets of bits ($vaddr_{dir}$, $vaddr_{table}$, $vaddr_{offset}$).

[2] http://uclid.eecs.berkeley.edu/s2w/

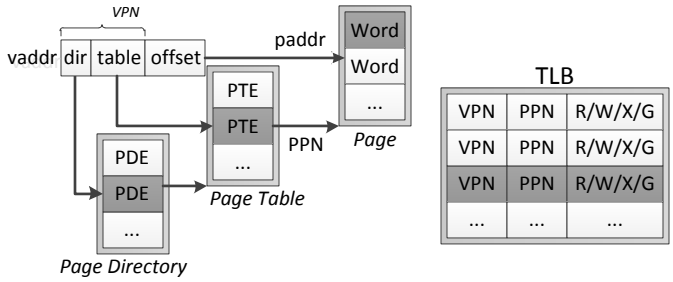

Figure 3.3: A page table walk.

On the left, we show a two-level page walk translating VPN to PPN addresses. The TLB caches VPN to PPN translations along with read/write/execute/global permission bits.

First, the $vaddr_{dir}$ bits index a page directory entry (PDE) within the page directory region. The PDE contents, along with the $vaddr_{table}$ bits, index into the page tables to retrieve a page table entry (PTE). The PTE contents identify a 4 KB physical page, and when concatenated with the 12 bit $vaddr_{offset}$, index a particular byte within this page. Since the above page walk includes two memory lookups, most x86 processors implement a TLB to cache VPN-to-PPN translations. The TLB also caches permission bits (r/w/x/g) checked during memory accesses. With this optimization, Bochs' address translation logic first checks its TLB for an entry describing the wanted VPN. If no such entry exists, Bochs performs a page walk to compute the corresponding PPN, and then stores that translation in its TLB for future access. We would like to prove that the optimized paging unit (with the TLB) is functionally equivalent to the original paging unit (without the TLB).

The Bochs TLB + page table system is modeled as a tuple $\mathcal{S}_{Bochs} = (\mathcal{I}, \mathcal{O}, \mathcal{V}, Init, \mathcal{A})$ where

- $\mathcal{I} = \{vaddr, data, pl, rwx, command\}$. *vaddr* is the virtual address to translate. *data* is used to update the page table memory. *pl* indicates the CPU's current privilege level (either user or supervisor mode). *rwx* indicates whether this memory access writes and/or executes this address.
- $\mathcal{O} = \{paddr\_TLB, pagefault\_TLB, paddr\_noTLB, pagefault\_noTLB\}$. *paddr_TLB* is the result of address translation with the TLB. *paddr_noTLB* is the result of address translation without the TLB. *pagefault_TLB* indicates a page fault occurred during translation with the TLB. The fault is caused by insufficient permissions. *pagefault_noTLB* indicates a page fault occurred during translation without the TLB.
- $\mathcal{V} = \{mem, TLB, legal\}$. *mem* is a 32-bit addressable memory containing both the page directory and page tables. *TLB* is an array (2^{10} entries in Bochs) of structs, where each struct is 160 bits wide and has 5 32-bit fields, including *vpn*, *ppn*, and access bits (*ab*). *legal* is a Boolean variable denoting whether the system reached the current state via a legal sequence of transitions.

Command	Modifies	Guard
write_pte	*mem*	`true`
write_pde	*mem*	`true`
translate	*TLB*	$\neg present \vee \neg permission$
set_cr3	*TLB*	`true`
invlpg	*TLB*	$TLB[vaddr_{table}].vpn_{31:12} = (vaddr_{dir} \circ vaddr_{table})$
invlpg_all	*TLB*	$TLB[vaddr_{table}].vpn_{31:22} = vaddr_{dir}$

Table 3.1: The allowable operations in our model of the Bochs TLB.

- $Init = (mem_0, TLB_0, \texttt{true})$, where $TLB_0[i].vpn :=$ 0xffffffff for all i and mem_0 is an uninterpreted function from 32-bit addresses to arbitrary 32-bit values. Initializing the TLB with the *vpn* field set to 0xffffffff in all entries marks it as empty. *legal* is initialized to `true`.
- $\mathcal{A}$: $\mathcal{V}$ evolves via operations *write_pde*, *write_pte*, *invlpg*, *invlpg_all*, *setcr3*, and *translate*, and the environment non-deterministically chooses one of these operations at each step. Table 3.1 describes each of these commands.

Each command is implemented in distinct functions within Bochs (src/cpu/paging.cc). Since Bochs executes on a single thread, we can safely model each function as an atomic operation, i.e., a single step in the state transition system. The commands *write_pde* and *write_pte* are used to update the page directory and page tables respectively, typically to modify access permissions or page mapping. *translate* performs address translation and assigns the result to variables in $\mathcal{O}$. Furthermore, if a page walk was deemed necessary, then *translate* updates a TLB entry with the results of that page walk. The *setcr3* command is used to switch to a new page table, typically during a context switch. If global pages are enabled, then *setcr3* flushes all non-global entries in the TLB. Otherwise if global pages are disabled, all TLB entries are flushed on a *setcr3* command. The x86 instruction *invlpg* flushes a specific TLB entry containing the translation for *vaddr*; *invlpg* is needed to invalidate the TLB entry following a write to the page table. *invlpg_all* atomically flushes all TLB entries that have $vaddr_{table}$ in their *vpn* (bits 31 to 22); *invlpg_all* is needed to invalidate a set of TLB entries following a write to the page directory.

We check equivalence of both the physical address and whether a page fault occurred. Since the x86 manual only guarantees cache coherency when the TLB is flushed properly, we only require equivalence on traces where each *write_pde* is followed by a *invlpg_all* and each *write_pte* is followed by *invlpg*. This constraint is enforced by *legal*, which is true only if the sequence of operations abides by these constraints. Any state that satisfies *legal* is guaranteed to be reachable from the initial state via a legal sequence of state transitions.

The property that we check is:

$$\begin{aligned}\Phi_{3.9} \doteq \forall v, p, r.\ (vaddr = v \wedge pl = p \wedge rwx = r) \rightarrow \\ legal \rightarrow ((pagefault\_TLB \Leftrightarrow pagefault\_noTLB) \wedge \\ (\neg pagefault\_noTLB \rightarrow (paddr\_noTLB = paddr\_TLB)))\end{aligned} \tag{3.9}$$

Induction

The one-step induction check consists of proving (3.10) and (3.11) using the UCLID verifier.

$$Init(\mathcal{V}_{Bochs}) \rightarrow \Phi_{3.9}(\mathcal{V}_{Bochs}) \tag{3.10}$$

$$\Phi_{3.9}(\mathcal{V}_{Bochs}) \wedge \mathcal{R}(\mathcal{V}_{Bochs}, \mathcal{V}'_{Bochs}) \rightarrow \Phi_{3.9}(\mathcal{V}'_{Bochs}) \tag{3.11}$$

Our initial state satisfies $\Phi_{3.9}$ because the TLB is initialized to be empty, thereby forcing both optimized and unoptimized designs to undergo the two-level page walk. However, one-step induction (check (3.11)) fails because the back-end SMT engine cannot solve the formula, which has a quantifier alternation. Consequently, we proceed on to the small and short world steps.

Small World

We syntactically derive the dependence set $\mathcal{U}_{\Phi_{3.9}}$ by traversing the expression graph of $\Phi_{3.9}$. After introducing a fresh 32-bit symbolic constant $v = (v_{dir}, v_{table}, v_{offset})$, the dependence set is

$$\begin{aligned}\mathcal{U}_{\Phi_{3.9}} = \{&legal, TLB[v_{table}], mem[cr3_{31:12} \circ v_{dir}], \\ &mem[mem[cr3_{31:12} \circ v_{dir}]_{31:12} \circ v_{table}]\}\end{aligned}$$

The last three expressions represent the TLB entry, page directory entry, and page table entry pointed to by v, respectively. (Here $cr3_{31:12}$ refers to the upper 20 bits of the cr3 control register and is modeled as a symbolic constant.) Our abstract model $\hat{\mathcal{S}}_{Bochs}$ precisely tracks only the variables in $\mathcal{U}_{\Phi_{3.9}}$; other state elements get updated with arbitrary values at each step.

Short World

We use the sub-sequence heuristic to find an upper bound on the diameter of $\hat{\mathcal{S}}_{Bochs}$. We find a bound of 9 steps. Finally, we perform bounded model checking for 9 steps, proving that all reachable states of $\hat{\mathcal{S}}_{Bochs} \| \mathcal{E}_{Bochs}$ satisfy $\Phi_{3.9}$. The sub-sequence check and BMC took roughly 45 minutes and 25 minutes, respectively.

Content Addressable Memory

While the TLB functions as a direct-mapped cache (each concrete logical address is associated with a single TLB entry), S^2W also applies to systems with set-associative caches and content addressable memories (CAMs). A CAM stores associations between keys and data. The key is typically stored as part of the data, and is used for comparison during lookups.

Figure 3.4 shows a system containing slow memory and a CAM-based cache. We would like to prove that a lookup in slow memory yields the same data as a lookup in the CAM-based cache, provided the data is present in the CAM. We model the CAM's state using a variable *cam* that maps a CAM index to its contents, a 65-bit vector containing fields *present*, *key*, and *data*. The $cam[i].present$ bit indicates whether the key $cam[i].key$ and data $cam[i].data$ are valid entries. If $cam[i].present$ is true, $cam[i].key$ and $cam[i].data$ contain the 32-bit key and 32-bit data stored at CAM index i. Memory is modeled as a variable *mem* mapping a 32-bit address to a 32-bit vector. That is, $mem[a]$ refers to the 32-bit data at address a.

We also maintain a state variable *map* that maps an address a to a 32-bit CAM index. *map* is updated when data is added or deleted from the CAM. A $read(addr)$ command checks the contents of the CAM at index $map[addr]$. If $cam[map[addr]].key = addr$ and $cam[map[addr]].present$ is true, then $\mathcal{S}_{CAM}$ assigns $cam[map[addr]].data$ to the output variable out_cam_{data} and `true`to the output variable $out\_cam_{present}$. Otherwise, $\mathcal{S}_{CAM}$ assigns `false`to $out\_cam_{present}$. *read* also assigns $mem[addr]$ to output variable out_mem_{data}. The $insert(addr, data)$ command checks *map* for an existing mapping of address *addr*. If $map[addr] \neq \perp$, then $\mathcal{S}_{CAM}$ updates the CAM at index $map[addr]$ with data *data* and key *addr*.[1] Otherwise, we arbitrarily choose a new location $arb[addr]$ to insert *data* and *addr*, and update $map[addr]$ with this new location. The $set(addr, data)$ command updates $mem[addr]$ with *data*, possibly making the CAM contents stale. The $reset(addr)$ command resynchronizes *cam* with *mem* at address *addr*, provided the CAM contains a valid entry for address *addr*. These commands are implemented using atomic operations, and they update *map*, *mem* and *cam* in parallel.

The CAM + memory system is modeled as a tuple $\mathcal{S}_{CAM} = (\mathcal{I}, \mathcal{O}, \mathcal{V}, Init, \mathcal{A})$ where

- $\mathcal{I} = \{addr, data, command\}$.
- $\mathcal{O} = \{out\_cam_{data}, out\_cam_{present}, out\_mem_{data}\}$.
- $\mathcal{V} = \{cam, mem, map, legal\}$. These are all modeled as bit-vector functions. *cam* returns a 65-bit vector: 1-bit *present*, 32-bit *data*, 32-bit *addr*. Both *mem* and *map* return a 32-bit vector. *legal* is a Boolean variable denoting whether the system legally reached the current state.

[1] Note that $\perp$ in our model equals 0x00000000; it is an acceptable design choice to not cache that address.

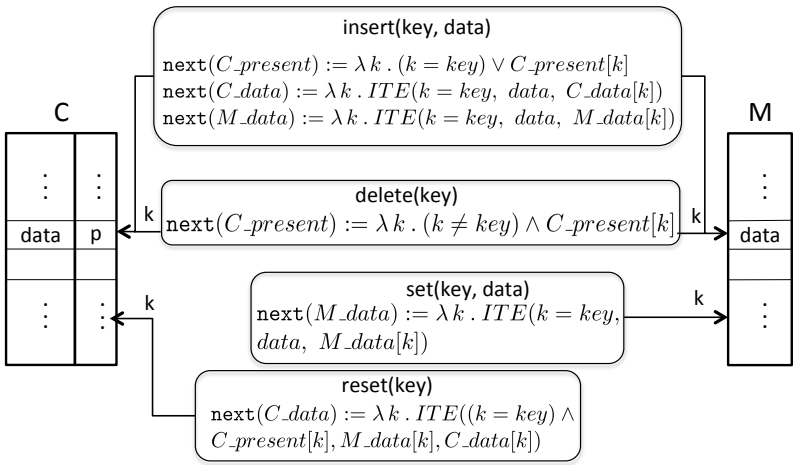

Figure 3.4: Our model of a slow memory and its CAM-based cache.

- $Init = (c_0, m_0, map_0, \texttt{true})$. $map_0[a] = \perp$ for all a, the present field of each CAM entry is initialized to `false`, memory is initialized to arbitrary values, and *legal* is initialized to `true`.
- $\mathcal{A}$: The state evolves via commands *insert*, *delete*, *set*, *reset* and *read*. The environment non-deterministically chooses one of these commands at each step. Figure 3.4 defines the state transition relation for each command.

The safety property $\Phi_{3.12}$ checks that the CAM and memory have the same data for all keys present in CAM. Note that the CAM only guarantees cache coherency if it is resynchronized with memory after each *set*. The state variable *legal* enforces this constraint: it is true if every *set* is followed by a *reset*.

$$\begin{aligned}\Phi_{3.12} \doteq \forall a.\ (addr = a) \rightarrow legal \rightarrow \\ (out\_cam_{present} \rightarrow (out\_cam_{data} = out\_mem_{data}))\end{aligned} \tag{3.12}$$

Since $\Phi_{3.12}$ is expressed over output variables, we check if a state s satisfies $\Phi_{3.12}$ by performing a *read* operation on state s with a fresh symbolic constant for a.

Induction

We first try one-step induction on this system to prove $\Phi_{3.12}$. The initial state satisfies $\Phi_{3.12}$ because both the CAM and *map* are empty. However, the inductive check fails because of the quantifier alternation, similar to the TLB case study. Hence, we continue onto the small world step of our approach.

Small World

We syntactically derive the dependence set by traversing the expression graph of $\Phi_{3.12}$. We introduce a fresh 32-bit symbolic constant a for the address that we precisely track in *map* and *mem*. We precisely track *legal*, $map[a]$, $mem[a]$, and $cam[map[a]]$.

$$\mathcal{U}_{\Phi_{3.12}} = \{legal, cam[map[a]], mem[a], map[a]\} \tag{3.13}$$

Our abstract model $\hat{\mathcal{S}}_{CAM}$ precisely tracks updates to only these variables.

Short World

Using the sub-sequence heuristic, we find an upper bound on the reachability diameter of 5 steps. Finally, we perform bounded model checking for 5 steps, proving that all reachable states of $\hat{\mathcal{S}}_{CAM} \| \mathcal{E}_{CAM}$ satisfy $\Phi_{3.12}$. The sub-sequence check and BMC takes about 15 seconds and 5 seconds respectively.

Shadow Paging

For our third case study, we model a shadow page table system. A hypervisor may use shadow page tables to assure address space separation between the guest and host. The guest page tables can be updated arbitrarily by the guest operating system, while the shadow page tables are updated only by the hypervisor. The hardware uses the shadow page tables for address translation. It is the hypervisor's responsibility to make sure the shadow page tables stay synchronized with the guest tables, while at the same time ensuring no translation will ever allow the guest to access memory outside its allocated sandbox. We model the synchronization process and verify that the physical address returned by translation never exceeds some constant limit, LIMIT.

Our shadow paging model (Figure 3.5) is as follows. There are two page table structures: guest and host. Each is a two-level structure: a page directory table (PDT) and a page table (PT). We refer to the guest and shadow page tables as gPDT, gPT and sPDT, sPT, respectively. Entries in the PDT have three fields: present (p), page-size-extension (pse), and address ($addr$). Entries in each nested PT have two fields: present (p) and address ($addr$).

Let $\mathcal{S}_{SP} = (\mathcal{I}, \mathcal{V}, Init, \mathcal{A})$ be the shadow paging model with

- $\mathcal{I} = \{i, j, command\}$. i and j index into the PDT and PT, respectively. *command* is one of *page-fault*, *inval-page*, *new-context*, or *adversary*.
- $\mathcal{V} = \{\text{gPDT}, \text{gPT}, \text{sPDT}, \text{sPT}, \text{LIMIT}\}$. gPDT, gPT, sPDT, and sPT are modeled as functions that map indices to bit-vectors. gPDT and sPDT return 34-bit vectors (1-bit p, 1-bit *pse*, 32-bit *addr*). gPT and sPT return 33-bit vectors (1-bit p, 32-bit *addr*). LIMIT is a constant 32-bit vector.

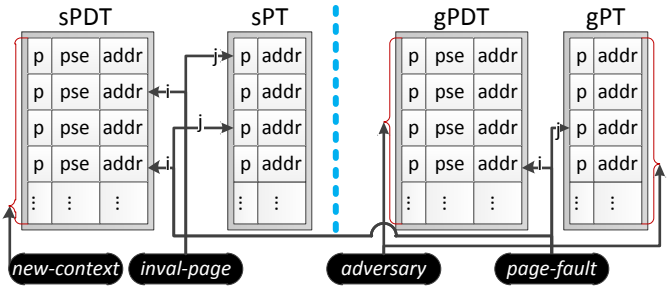

Figure 3.5: An illustration of the shadow page table model.

- $Init = (\text{sPDT}_0, \text{sPT}_0, \text{gPDT}_0, \text{gPT}_0)$. sPDT and sPT are both initialized with the p bit cleared in all entries. gPDT and gPT are initialized to arbitrary values.
- $\mathcal{A}$. The four commands update state in the following way: *page-fault* synchronizes the shadow tables with the guest tables. *inval-page* conditionally invalidates (zeros out) entries in sPDT and sPT. *new-context* unconditionally invalidates entries in sPDT. *adversary* writes to gPDT and gPT. The assignments to gPDT, gPT, sPDT, and sPT are summarized in Table 3.2.

Command	Modifies	Guard
page-fault(i,j)	sPDT$[i]$	gPDT$[i].pse$ $\wedge$ gPDT$[i].p\wedge$ (gPDT$[i].addr <$ LIMIT)
	sPDT$[i]$	gPDT$[i].pse$ $\wedge \neg$(gPDT$[i].p\wedge$ (gPDT$[i].addr <$ LIMIT))
	sPDT$[i]$, sPT$[j]$	$\neg$gPDT$[i].pse$ $\wedge$ gPDT$[i].p\wedge$ (gPDT$[i].addr <$ LIMIT)$\wedge$ gPT$[j].p \wedge$ (gPT$[j].addr <$ LIMIT)
	sPDT$[i]$	$\neg$gPDT$[i].pse$ $\wedge$ gPDT$[i].p\wedge$ (gPDT$[i].addr <$ LIMIT)$\wedge$ $\neg$(gPT$[j].p \wedge$ (gPT$[j].addr <$ LIMIT))$\wedge$ $\neg$(sPDT$[i].p \wedge \neg$sPDT$[i].pse$)
	sPDT$[i]$	$\neg$gPDT$[i].pse$ $\wedge \neg$(gPDT$[i].p\wedge$ (gPDT$[i].addr <$ LIMIT))
	sPT$[j]$	$\neg$gPDT$[i].pse$ $\wedge$ gPDT$[i].p\wedge$ (gPDT$[i].addr <$ LIMIT)$\wedge$ $\neg$(gPT$[j].p \wedge$ (gPT$[j].addr <$ LIMIT))$\wedge$ (sPDT$[i].p \wedge \neg$sPDT$[i].pse$
inval-page(i,j)	sPDT$[i]$	(sPDT$[i].p \wedge \neg$gPDT$[i].p)\vee$ (sPDT$[i].p \wedge$ gPDT$[i].p\wedge$ (sPDT$[i].pse \vee$ gPDT$[i].pse$))
	sPT$[j]$	sPDT$[i].p \wedge$ gPDT$[i].p\wedge$ $\neg$gPDT$[i].pse \wedge \neg$sPDT$(i).pse$
new-context	gPDT	`true`
adversary	gPDT gPT	`true` `true`

Table 3.2: Next-state assignments for the shadow paging model.

Our model is based on the ShadowVisor model [102], but has been extended to introduce

pointers. The safety properties we verify are similar to those of ShadowVisor. We verify that a translation using the shadow page tables will never return an address above a fixed limit.

$$\Phi_{3.14} = \forall i.\ (\text{sPDT}[i].p \wedge \text{sPDT}[i].pse) \rightarrow \text{sPDT}[i].addr < \text{LIMIT} \tag{3.14}$$

$$\Phi_{3.15} = \forall i, j.\ (\text{sPDT}[i].p \wedge \neg\text{sPDT}[i].pse \wedge \text{sPT}[j].p) \rightarrow \text{sPT}[j].addr < \text{LIMIT} \tag{3.15}$$

Induction

The property $\mathbf{G}\ \Phi_{3.14}$ is proven by induction. However, one-step induction fails to prove $\mathbf{G}\ \Phi_{3.15}$. If sPT$[j]$ is marked as present, but has an address greater than LIMIT, $\Phi_{3.15}$ is still true as long as sPDT$[i]$ is marked not present. From that state, it is possible to update sPDT$[i]$ to present without updating sPT$[j]$, so that in the next state the sPDT$[i]$ and sPT$[j]$ entry are both marked present and the data in the sPT$[j]$ entry is greater than LIMIT, violating $\Phi_{3.15}$. Therefore, we move on to the small and short world steps for $\mathbf{G}\ \Phi_{3.15}$.

Small World

The properties are concerned with a single entry i in sPDT and a single entry j in sPT. Therefore we use a fresh symbolic constant to choose an arbitrary entry from each. In particular, our conditional dependency set is $\mathcal{U}_{\phi_{3.15}} = \{\text{sPDT}[a_i], \text{sPT}[a_j]\}$. In our abstract symbolic transition system, $\hat{\mathcal{S}}_{SP}$, we track precisely only the state in $\mathcal{U}_{\phi_{3.15}}$. a_i, a_j stay constant.

Short World

The sub-sequence short-world heuristic does not work for page tables, because page table entry updates can depend on previous writes to the entry or to other entries; it is not always possible to drop one step of a trace to achieve an equivalent final state. Instead, we use gadgets to find and prove the short world. We manually construct a universal gadget set to prove the length of the diameter of $\hat{\mathcal{S}}_{SP}$.

To build the gadgets we case split on the possible end-state valuations for the variables in $\mathcal{U}_{\phi_{3.15}}$, and for each, determine how to get there from a valid starting state. The model has only four commands and it was usually obvious which commands were needed to get to a particular state. The gadgets must also specify the parameters (i, j) to the command, and, in the case of the *adversary* command, the value that gets written to the guest tables. Figuring out the correct parameters to use for each command was more difficult. In this case, the parameters were always either $i := a_i$ or $i := a_i'$ (where $a_i' \neq a_i$ is arbitrarily

chosen), and similarly for j. The *adversary* data that gets written to the guest tables was a combination of the *addr* field of the (symbolic) end-state valuation we were trying to achieve and a particular value for the p and pse bits, which we chose according to the particular gadget we were building.

We needed a total of thirteen gadgets, each four commands or less, to prove the short world has length four. We then ran BMC on $\hat{\mathcal{S}}_{SP}$ for four steps and verified the property held at each step. The verification of the short world took less than a minute; BMC took approximately five seconds.

Other Hypervisor Models

We applied our abstraction technique to three additional hypervisor models. All were verified using one-step induction, each within 5 seconds.

SecVisor

SecVisor [29, 103] is a small hypervisor that supports a single guest OS. It virtualizes the memory management unit by implementing shadow page tables and synchronizing them with the guest page tables. The model assumes an adversary can write arbitrary values to the guest page tables. SecVisor aims to execute only approved code in kernel mode; therefore, the page-table synchronization must prevent adversary-provided code from having execute permissions while in kernel mode. We verified the security property using one-step induction on a model given by Franklin et al. [103].

sHype

sHype [103] is an access control system used by the Xen [51] hypervisor. Based on the Chinese Wall policy, it establishes "conflict of interest" classes and guarantees each virtual machine will never access two pieces of data from the same conflict of interest class. We verified the security property using a model presented by Franklin et al. [103].

ShadowVisor

ShadowVisor [102] served as the starting point for our shadow page table model. It models the page tables of a simple hypervisor that assures address space separation between guest and host by maintaining separate guest and shadow page tables. Like our shadow page table model, ShadowVisor guarantees that if an address is marked as present, it will never exceed a certain fixed limit. We model ShadowVisor in our modeling language and use one-step induction to verify the property.

3.5 Related Work

Verification of infinite-state or parametrized systems has been well-studied. Here we present the most closely-related work.

Franklin et al. [102,103] present a small-model approach to verifying systems with parametrized data structures, specifically, arrays. In essence, they present a formal language such that if the system can be modeled in their language, then a small-model theorem applies, stating that the unbounded arrays can be reduced to arrays with one designated element alone. Finite-state model checking can then be employed on the resulting system. While this approach is elegant, it does make a trade-off in expressiveness. Our modeling language is more expressive, allowing us to model the Bochs TLB, CAM, and shadow paging examples which cannot be modeled in their language. And, our approach is just different: we compute an abstraction based on localization within the large data structures, and we use bounded model checking.

The use of inductive invariant checking is common in this problem domain. The method of *invisible invariants* [115,116] is an example of an inductive verification technique applied to systems of N identical finite-state processes. The core idea in this method is to generalize from the reachable states of a small number of processes into a quantified inductive assertion of the form $\forall i.\phi(i)$, where the index i ranges over process IDs. Namjoshi [104] also discusses the so-called *cutoff method*, which is a small-model approach for such systems of parametrized processes, drawing connections between inductive methods, small-model approaches and compositional reasoning. In general, these approaches are not easily applied to our examples since they are not naturally decomposed into a system of N identical finite-state processes. Instead, in our problem domain, the number of interacting processes is usually finite and small, but the shared data structures are large and complicated.

Abstraction-based approaches have also been presented for infinite-state or parametrized systems similar to those studied in this work. Lahiri and Bryant [105] presented an approach for verifying universally quantified invariants on parametrized systems using predicate abstraction. While predicate abstraction can be quite effective for verifying control-related properties, especially when one can guess suitable predicates, it is not suitable for verifying equivalence of two code versions (such as in the Bochs TLB case study), which is a highly data-dependent property. McMillan [117] presents a semi-automatic approach to compositional reasoning using an abstraction similar to ours. Given a system with large arrays, he uses a form of localization abstraction guided by case splitting performed by the user to only model a few entries in the arrays precisely. All other entries are allowed to be updated by an arbitrary value $\perp$. This abstraction is used to compute a finite state abstract model on which reachability analysis is performed. In contrast, our method computes an abstraction based on index terms derived from the property, and uses a BMC-based approach to verify the system. Bjesse [106] describes an automatic approach for verifying sequential circuits with large memories, if the memories are "remodelable" as formally defined by Bjesse.

The work takes a small-world approach, transforming an initial netlist into another with memories with precise updates only to a small number of entries in memories, and uses counter-example-guided abstraction refinement. Our work does not require the remodelable restriction. German [107] presents a novel approach for constructing sound and complete abstractions for similar systems. The approach is based on performing static analysis on the system model, and, in contrast to Bjesse's work, can handle unbounded delays between the time the array is read and when the read value propagates to the output. While the approach is completely automatic, it cannot handle certain data structures such as CAMs, in which a read, in principle, requires scanning the entire array. Our approach, while not always automatic, does handle structures such as CAMs (see Section 3.4).

Efficient memory modeling is a technique used for bounded verification problems, either symbolic simulation (e.g., [118]) or bounded model checking (e.g., [119]). In contrast, our approach focuses on unbounded verification based on abstraction and a sound application of BMC based on heuristics to find the reachability diameter.

3.6 Conclusion

In this chapter, we presented S^2W, a new approach to verifying systems with large or unbounded data structures that combines induction and abstraction-based model checking. We presented experimental results on several examples of emulators and hypervisors and demonstrated that this technique can be used to handle the large data structures present in virtualization software. In the next chapter we investigate how to validate the types of models necessary in this, or any, model checking-based approach.

Chapter 4

Model Validation

Modeling is the crucial first step in formal verification techniques based on model checking. A model may be constructed manually from source code or extracted automatically by a tool. In either case, the inherent weakness of this approach is that successful verification means the property has been proven true of the model, but not necessarily of the original system. This weakness is fairly clear for manually constructed models, but it is important to note that it also holds for tools operating directly on source code. Such software model checkers construct and internally maintain a model of the code, on which the analysis is actually performed. Moreover, in order to scale to large code bases, such tools often require human guidance in the form of code annotations or modifications. Thus, in these approaches too, one implicitly trusts that the model is correct: if verification of the model succeeds, one assumes the verification would hold for the original system as well.

In this work we present a framework for validating that assumption. We formalize the process of validating a model against the original source code implementation of the system to prove that a property proven true of the model is also true of the system. We concentrate solely on validating models that will be used for the verification of safety properties. Our framework is targeted towards systems implemented in imperative languages such as C/C++. A special focus is on systems software that exports several "logical operations" that client programs can use (or misuse). Examples include a virtual machine monitor that provides memory isolation and other services to operating systems hosted on it, or an FTP server implementing several commands. Such systems are best understood as being logically concurrent, even if they are single-threaded. Each logical operation (or set of operations) has an associated collection of data that it operates upon. Therefore, the corresponding models are usually best specified as transition systems with several logical operations that can be interleaved in a specified manner to operate on their associated data structures based on client inputs. The complexity of the associated data structures often necessitates the use of abstraction using first order logic with suitable background theories.

In order to be sound, in the general case, model validation must show that all states reachable

in the original system are also reachable in the model. However, often a large system will have many states that are equivalent with respect to the property to be verified. When building a model, modelers often take advantage of this fact and do not bother to model irrelevant portions of the system. For example, the Bochs CPU Emulator [54] is a large system comprising over 400 C/C++ files, in addition to several files in other languages, and requires more than 200 KLOC.[1] Building a model of the entire system would be a daunting, and possibly unnecessary task. A property about the correctness of address translation, for example, might require only a small handful of the modules be included in the model. Modules that manage peripheral devices or handle the instruction fetch and decode functions of the CPU might safely be left out of the model. The first phase of model validation, therefore, is to show that in constructing the model, the modelers did the pruning correctly: there is no portion of the system that is relevant to the property under consideration, but that was left out of the model. We do that by showing, for all variables relevant to the property, there is no program code that writes the variable, but that was pruned out during modeling.

The second phase of model validation is to show that the model is an overapproximate representation, with respect to the property under consideration, of the corresponding portion of the original system. We do that by showing that the model simulates this portion of the code. The simulation check is performed by checking that each logical operation in the code is overapproximated by a corresponding logical operation in the model. This check is formulated as checking validity of a logical formula and discharged using an SMT solver [120].

We propose the following work flow (see Figure 4.1):

1. *System Pruning*: Before the model is built, the modelers first prune out irrelevant code. Given a system $\mathcal{S}$ and a property Φ, the relevant portions of the code, $\mathcal{F}_{\mathcal{P}}$, are identified. This is typically a manual step that requires some domain expertise.
2. *Data-Centric Model Validation (DMV)*: In the first phase of model validation, the pruning done in the previous step is validated. If DMV finds that $\mathcal{F}_{\mathcal{P}}$ is missing some relevant code, the process returns to Step 1.
3. *Model Construction*: Given $\mathcal{F}_{\mathcal{P}}$, modelers build a formal model $\mathcal{M}$ of the pruned system.
4. *Operation-Centric Model Validation (OMV)*: In the second phase of model validation, $\mathcal{M}$ is validated as a safe abstraction of $\mathcal{F}_{\mathcal{P}}$ with respect to Φ. If OMV finds any discrepancies, the process returns to the previous step.
5. *Verification*: Once model validation is done, model checking-based verification can be completed.

In this work we focus on Steps 2 and 4 of the work flow; we assume the pruning and modeling are done manually, and that any one of a number of standard model checking techniques can be used in Step 5. In our theoretical framework, the DMV and OMV steps are sound. However, our current implementation does not share this property. We use

[1] Calculated for Bochs 2.6.2 using SLOCCount (http://www.dwheeler.com/sloccount/).

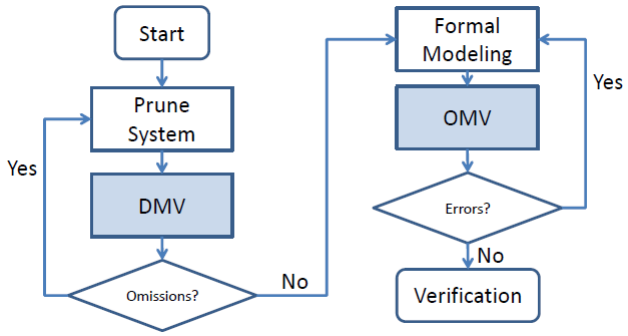

Figure 4.1: Workflow for model checking-based verification.

Model validation comprises the DMV and OMV processes, which are shown shaded in the figure.

symbolic execution [121] and SMT solving [120] to implement the OMV and DMV steps. The former is used to explore all possible feasible paths through the code, and the latter is used to check the validity of queries generated in the DMV and OMV steps. Due to limitations of current symbolic execution tools, the DMV step will, in some cases, fail to catch portions of $\mathcal{S}$ that were omitted from $\mathcal{F}_{\mathcal{P}}$. We rely on the domain knowledge and expertise of the modelers to identify the relevant portions of the system to model and use the DMV step to help find any missed code. While our implementation does not guarantee that all relevant code will be identified during this step, it does give the modelers increased confidence that no relevant portions were missed.

Our implementation currently uses KLEE [122] for symbolic execution, the UCLID [108] language for building our models, and the UCLID decision procedure for proving the SMT queries that check validation. The methodology can be made to work with other tools as well.

4.1 Running Example

We present here an example snippet of code and the corresponding model, which we will use in subsequent sections as we describe our algorithm. The code snippet, shown in Figure 4.2, is based on code taken from the Bochs CPU emulator [54]. It is a simple `if`, `then`, `else` statement that tests, for memory pages marked as supervisor-only, whether the CPU has a current privilege level (CPL) of 3. If it does, a page fault will be raised (the page-fault flag is set to 1), indicating a protection check has failed for this supervisor-only page. Otherwise, the page-fault flag is set to 0. After a successful check for a page fault (not shown), the access and dirty bits of the page table entry are updated. In this example `curr_privilege_level` and `page_fault` are both integer variables.

The modeler asserts that the `update_access_dirty` function is irrelevant as far as the CPL

```
if (curr_privilege_level == 3)
    page_fault = 1;
else page_fault = 0;
...
update_access_dirty(...);
```

Code.

```
page_fault := (cpl[0] & cpl[1]);
```

Model.

Figure 4.2: Running example.

A snippet of code taken from the Bochs CPU emulator and its corresponding hand-written model. In the model, `curr_privilege_level` is abbreviated to `cpl`.

is concerned, so she excludes it from validation and from modeling.

The corresponding model is also shown in Figure 4.2. In the model, if bits 0 and 1 of the variable storing the current privilege are both set, the page-fault flag is set to 1, otherwise, the page-fault flag is set to 0. In the model, `cpl` is modeled as a 32-bit bit-vector, and `page_fault` is modeled as a 1-bit bit-vector.

4.2 Theoretical Formulation and Approach

Notation and Background

We introduce here some terminology and background material used in the rest of this chapter.

Program Throughout this chapter, we use "system" or $\mathcal{S}$ to refer to the original software system to be verified; Φ is the property to be proven or disproven during verification.

The system $\mathcal{S}$ is represented as a tuple $\mathcal{S} = (\mathcal{I}_\mathcal{S}, \mathcal{V}_\mathcal{S}, \mathit{Init}_\mathcal{S}, \mathcal{C}_\mathcal{S})$, where

- $\mathcal{I}_\mathcal{S}$ is a finite set of input variables;
- $\mathcal{V}_\mathcal{S}$ is a finite set of state variables;
- $\mathit{Init}_\mathcal{S}$ is a predicate characterizing the set of initial valuations to variables in $\mathcal{I}_\mathcal{S}$ and $\mathcal{V}_\mathcal{S}$;
- $\mathcal{C}_\mathcal{S}$ is the code for the program, written in an imperative programming language such as C or C++, and describing how the system variables are updated.

The input variables, $\mathcal{I}_\mathcal{S}$, are the read-only variables of the system. The state variables, $\mathcal{V}_\mathcal{S}$, are all other variables of the program, including any global variables and return variables.

We make no assumptions about $\mathcal{C}_\mathcal{S}$ except the availability of a function σ that maps $\mathcal{C}_\mathcal{S}$ to a transition relation between pairs of system states, given a notion of a step. In other words, σ gives relational semantics to $\mathcal{S}$.

We use the term "annotated system" or $\mathcal{A}$ to mean a representation of $\mathcal{S}$ that highlights particular code fragments as being relevant to the verification task at hand. More formally, we define $\mathcal{A}$ as a tuple: $\mathcal{A} = (\mathcal{I}_\mathcal{A}, \mathcal{V}_\mathcal{A}, \mathit{Init}_\mathcal{A}, \mathcal{F}_\mathcal{A})$ where

- $\mathcal{I}_\mathcal{A}$ is a finite set of input variables: $\mathcal{I}_\mathcal{A} = \mathcal{I}_\mathcal{S}$;
- $\mathcal{V}_\mathcal{A}$ is a finite set of state variables: $\mathcal{V}_\mathcal{A} = \mathcal{V}_\mathcal{S}$;
- $\mathit{Init}_\mathcal{A}$ is a predicate characterizing the set of initial valuations to variables in $\mathcal{I}_\mathcal{A}$ and $\mathcal{V}_\mathcal{A}$;
- $\mathcal{F}_\mathcal{A} = \{f_1, f_2, \ldots, f_N, f_{\text{misc}}, f_{\text{orc}}\}$ is a finite set of code fragments in $\mathcal{C}_\mathcal{S}$, where $f_1, f_2, \ldots, f_N$ are code fragments that capture the relevant parts of $\mathcal{C}_\mathcal{S}$, f_{misc}, which is disjoint from $f_1, f_2, \ldots, f_N$, represents the code in $\mathcal{C}_\mathcal{S}$ deemed irrelevant, and f_{orc} is the code that orchestrates how program execution interleaves between the code fragments $f_1, f_2, \ldots, f_N, f_{\text{misc}}$.

For the rest of this discussion, we assume the code fragments $f_1, f_2, \ldots, f_N, f_{\text{misc}}$ to be terminating.[2] We do not make this assumption about the orchestration code f_{orc} since in general, for reactive systems like an operating system or an emulator, f_{orc} is designed to run forever. Often, the code fragment f_i $(1 \leq i \leq N)$ is a single C/C++ function in $\mathcal{C}_\mathcal{S}$, but in general we only require that each code fragment have clearly specified entry and exit points. We also require that each of $f_1, f_2, \ldots, f_N$ executes atomically.

The code f_{orc} dictates how the code fragments $f_1, f_2, \ldots, f_N, f_{\text{misc}}$ are composed together. For example, f_{orc} might simply be the sequential composition of two code fragments. A more complicated orchestrator f_{orc} might iterate over a "while(1)" loop repeatedly selecting a code fragment to execute based, for example, on a scheduling policy or on a sequence of external inputs. An example of the latter is a CPU emulator that repeatedly emulates instructions within a loop, where each instruction type has an associated function that emulates it. The form of the orchestrator depends heavily on the type of system under verification.

The function σ applies to $f_1, f_2, \ldots, f_N, f_{\text{misc}}$ in exactly the same way as it applies to $\mathcal{C}_\mathcal{S}$ – it characterizes the code fragments in terms of their underlying transition relation, given the notion of a step. In this work, since we assume that $f_1, f_2, \ldots, f_N, f_{\text{misc}}$ are terminating, the notion of a step will be taken to be a single execution of a code fragment. Thus, $\sigma(f_i)$ is the set of (pre, post) state pairs of f_i. We will often abbreviate $\sigma(f_i)$ by δ_i.

From $\mathcal{A}$, we can create a program $\mathcal{P}$ by dropping f_{misc} from the set $\mathcal{F}_\mathcal{A}$ and treating any invocation of f_{misc} from f_{orc} as a no-op (stuttering step). We define $\mathcal{P}$ as a tuple analogously to $\mathcal{A}$: $\mathcal{P} = (\mathcal{I}_\mathcal{P}, \mathcal{V}_\mathcal{P}, \mathit{Init}_\mathcal{P}, \mathcal{F}_\mathcal{P})$ where

- $\mathcal{I}_\mathcal{P}$ is a finite set of input variables: $\mathcal{I}_\mathcal{P} \subseteq \mathcal{I}_\mathcal{A} \cup \mathcal{V}_\mathcal{A}$;
- $\mathcal{V}_\mathcal{P}$ is a finite set of state variables: $\mathcal{V}_\mathcal{P} \subseteq \mathcal{V}_\mathcal{A}$;
- $\mathit{Init}_\mathcal{P}$ is a predicate characterizing the set of initial valuations to variables in $\mathcal{I}_\mathcal{P}$ and $\mathcal{V}_\mathcal{P}$;
- $\mathcal{F}_\mathcal{P} = \{f_1, f_2, \ldots, f_N, f_{\text{orc}}\}$ is a finite set of code fragments. The f_i and f_{orc} are defined as

[2] Note that it may be possible to lift the assumption about f_{misc}, the code deemed irrelevant to the verification task, provided that the technique for checking this irrelevance can handle non-terminating code. However, we retain this assumption in the present discussion.

in $\mathcal{A}$.

The transition relation of the overall program $\mathcal{P}$ is determined from those of its code fragments by the form of composition, i.e., by f_{orc}; we will denote it as $\delta^{\mathcal{P}}$.

As an example, consider the program in Figure 4.2. This program is modeled as the tuple $\mathcal{P} = (\mathcal{I}_{\mathcal{P}}, \mathcal{V}_{\mathcal{P}}, \mathit{Init}_{\mathcal{P}}, \mathcal{F}_{\mathcal{P}})$, where $\mathcal{I}_{\mathcal{P}} = \{\mathit{current\_privilege\_level}\}$, $\mathcal{V}_{\mathcal{P}} = \{\mathit{page\_fault}\}$, $\mathit{Init}_{\mathcal{P}} =$ `true` (allowing arbitrary values to *page_fault* and *curr_privilege_level*), and $\mathcal{F}_{\mathcal{P}} = \{f_1, f_{\text{orc}}\}$ where f_1 includes all the code except the function `update_access_dirty` and f_{orc} is the sequential composition $f_1; f_{\text{misc}}$. In the corresponding $\mathcal{A}$, f_{misc} was defined to include only `update_access_dirty`.

Symbolic Execution One approach to computing the relational semantics of a code fragment f is to enumerate its paths, using symbolic execution [121] to compute path conditions which can then be combined to form the transition relation $\sigma(f)$.

A program is symbolically executed in the following way. Initially, the variables in $\mathcal{I}_{\mathcal{P}}$ and $\mathcal{V}_{\mathcal{P}}$ are each represented as a symbolic value. Each variable starts with a fresh, unconstrained symbolic value. As the program executes, operations are computed symbolically. At the first conditional branch, execution can continue down one of two paths. The symbolic execution engine forks execution, and for each path, creates a new *path condition*, which is a predicate $P(\mathcal{I}_{\mathcal{P}}, \mathcal{V}_{\mathcal{P}})$ on the input and state variables that is true along that path. At each subsequent conditional branch and execution fork, the path condition for the current path of execution is updated with the new predicate. If pc is the current path condition and a conditional branch creates the two predicates $P, \neg P$, then after forking execution, the two new path conditions will be $pc_1 = pc \wedge P$ and $pc_2 = pc \wedge \neg P$. After execution has completed, for every path explored, there is a path condition pc, which is a conjunction of predicates on the input and state variables. Restricting $\mathit{Init}_{\mathcal{P}}$ to values that satisfy the path condition and executing the program will always force execution down the same path. Along with each pc we can also take note of the final value of the state variables of the program, $\mathcal{V}_{\mathcal{P}}{}'$. These values may be concrete or symbolic or some combination of the two. Any subsequent execution of the program along the path described by pc will produce the output or next-state consistent with $\mathcal{V}_{\mathcal{P}}{}'$.

Once the symbolic execution engine has explored all possible paths through the program fragment, we have a set $R = \{(pc, \mathcal{V}_{\mathcal{P}}{}')\}$ of pairs of path conditions and their corresponding output state. This set effectively describes the input–output relation of the program.

Going back to the example in Figure 4.2, note that there are two possible paths through the code with corresponding path conditions

$$pc_1 : \mathit{curr\_privilege\_level} = 3$$
$$pc_2 : \mathit{curr\_privilege\_level} \neq 3.$$

The corresponding next-states for each path are:

$$\mathcal{V}'_{\mathcal{P}_1} : \mathit{page\_fault} = 1$$
$$\mathcal{V}'_{\mathcal{P}_2} : \mathit{page\_fault} = 0.$$

Model We now formalize the class of models considered in this work. Our models are constructed as transition systems where state variables have one of three possible types: Boolean, bit-vector, and memory. The last type can be used to model arrays or various data structures. The grammar for expressions in our modeling language is the same as described in Section 3.2 and shown in Figure 3.2.

A formal model is represented as a tuple $\mathcal{M} = (\mathcal{I}_\mathcal{M}, \mathcal{V}_\mathcal{M}, \mathit{Init}_\mathcal{M}, \mathcal{O}_\mathcal{M})$ where

- $\mathcal{I}_\mathcal{M}$ is a finite set of input variables;
- $\mathcal{V}_\mathcal{M}$ is a finite set of state variables;
- $\mathit{Init}_\mathcal{M}$ is a predicate characterizing the set of initial valuations to variables in $\mathcal{I}_\mathcal{M}$ and $\mathcal{V}_\mathcal{M}$;
- $\mathcal{O}_\mathcal{M} = \{op_1, op_2, \ldots, op_N, \mathit{main}\}$ is a finite set of operations that determine how the state variables evolve, where *main* is a specially designated "top-level" operation that determines how the remaining operations are composed together. $\mathcal{O}_\mathcal{M}$ defines the transition relation $\delta^\mathcal{M}$ of the model, as described below.

Each operation is a finite set of assignments to variables in $\mathcal{V}_\mathcal{M}$. Assignments define how state variables are updated in a single step of the transition system. As in Section 3.2, a next-state assignment α updates a state variable and is a rule with one of the following forms:

$$\texttt{next}(x) := e,$$
$$\texttt{next}(x) := \{e_1, e_2, \ldots, e_n\}, \text{or}$$
$$\texttt{next}(x) := *$$

where x is a signal in $\mathcal{V}_\mathcal{M}$, $e, e_1, e_2, \ldots, e_n$ are expressions in the grammar of Figure 3.2 that are a function of $\mathcal{V}_\mathcal{M} \cup \mathcal{I}_\mathcal{M}$, and "$*$" is a wildcard that is translated at each transition into a fresh symbolic constant of the appropriate type. The curly braces express non-deterministic choice.

The transition relation corresponding to an operation op_i, denoted $\delta_i^\mathcal{M}$, is computed as $\delta_i^\mathcal{M} = \bigwedge_{\alpha \in op_i} r(\alpha)$, where

$$r(\texttt{next}(x) := e) \doteq (x' = e);$$
$$r(\texttt{next}(x) := \{e_1, e_2, \ldots, e_n\}) \doteq \bigvee_{i=1}^{n} (x' = e_i); \text{ and}$$
$$r(\texttt{next}(x) := *) \doteq (x' = *);$$

and x' denotes the next-state version of variable x.

The overall transition relation of the model $\mathcal{M}$ is determined by the form of composition expressed in *main*. For example, if *main* defines $\{op_1, op_2, \ldots, op_N\}$ to be composed asynchronously, then the overall transition relation $\delta^{\mathcal{M}} = \bigvee_{i=1}^{N} \delta_i^{\mathcal{M}}$.

As an example, consider the model in Figure 4.2. It is expressed as the tuple ($\mathcal{I}_{\mathcal{M}}$, $\mathcal{V}_{\mathcal{M}}$, $Init_{\mathcal{M}}$, $\mathcal{O}_{\mathcal{M}}$), where $\mathcal{I}_{\mathcal{M}} = \{cpl\}$, $\mathcal{V}_{\mathcal{M}} = \{page\_fault\}$, $Init_{\mathcal{M}}$ is `true` (allowing arbitrary values to *cpl* and *page_fault*), and $\mathcal{O}_{\mathcal{M}}$ has a single operation *op* which in turn contains only the following next-state assignment:

$$\texttt{next}(page\_fault) := ITE(cpl[0] = 1 \wedge cpl[1] = 1, 1, 0)$$

Environment The environment $\mathcal{E}$ that provides inputs to the program is similarly modeled as a transition system, where the input variables in $\mathcal{M}$ are the state variables of $\mathcal{E}$. The final model that is to be verified is the composition of $\mathcal{M}$ and $\mathcal{E}$, written $\mathcal{M}\|\mathcal{E}$. The form of the composition depends on the context; both synchronous and asynchronous compositions are possible. However, for this work, and in all of our examples, the environment $\mathcal{E}$ is stateless, generating completely arbitrary inputs to $\mathcal{M}$ at each step.

Transition Systems and Simulation Notice that the original software system $\mathcal{S}$, the program $\mathcal{P}$, and its model $\mathcal{M}$ are all transition systems with the same basic form. Once composed with an environment model $\mathcal{E}$, the resulting transition system has the form $\mathcal{T}$ = ($\mathcal{V}$, *Init*, δ), where $\mathcal{V}$ are the state variables, *Init* is the initial condition, and δ is the transition relation. A state s of $\mathcal{T}$ is a valuation to the variables in $\mathcal{V}$.

Given a transition system $\mathcal{T}$ and a set of variables $\mathcal{V}_L \subseteq \mathcal{V}$, each state s of $\mathcal{T}$ can be labeled with a valuation to variables in $\mathcal{V}_L$. We denote this labeling as $\mathcal{L}(s)$.

Given two transition systems $\mathcal{T}_1$ and $\mathcal{T}_2$ sharing a common labeling function $\mathcal{L}$, $\mathcal{T}_1$ is said to *simulate* $\mathcal{T}_2$ if there exists a simulation relation $\mathcal{H}$ relating states in $\mathcal{T}_1$ and $\mathcal{T}_2$ with the following properties:

(i) $\forall s_1, s_2.\mathcal{H}(s_1, s_2) \implies \mathcal{L}(s_1) = \mathcal{L}(s_2)$
(ii) $\forall s_1, s_2, s_2'.[\mathcal{H}(s_1, s_2) \wedge \delta_2(s_2, s_2')] \implies \exists s_1'.\delta_1(s_1, s_1') \wedge \mathcal{H}(s_1', s_2')$
(iii) $\forall s_2.Init_2(s_2) \implies (\exists s_1.Init_1(s_1) \wedge \mathcal{H}(s_1, s_2))$

We write "$\mathcal{T}_1$ simulates $\mathcal{T}_2$" as $\mathcal{T}_2 \preceq \mathcal{T}_1$.

A slight variation is the notion of stuttering simulation. We say that $\mathcal{T}_1$ *stutter simulates* $\mathcal{T}_2$ if there exists a relation $\mathcal{H}_{\text{st}}$ between states in $\mathcal{T}_1$ and $\mathcal{T}_2$ where the second condition above is modified as follows:

(ii) $\forall s_1, s_2, s_2'.[\mathcal{H}(s_1, s_2) \wedge \delta_2(s_2, s_2')] \implies [\mathcal{H}(s_1, s_2') \vee \exists s_1' \delta_1(s_1, s_1') \wedge \mathcal{H}(s_1', s_2')]$.[3]

[3]Note that this definition is a slight variant of the standard one which allows any finite number of

We write "$\mathcal{T}_1$ stutter simulates $\mathcal{T}_2$" as $\mathcal{T}_2 \preceq_{\text{st}} \mathcal{T}_1$.

Problem Definition and Approach

Models are always constructed based on the properties to be verified. Therefore, a model must be validated with respect to a specific property Φ.

In this work, we focus on the verification of temporal safety properties of the form $\mathbf{G}\phi$, where $\mathbf{G}$ is the temporal operator "always" and ϕ is a state predicate, i.e., a Boolean expression over state variables with no temporal operators.

The overall model validation problem definition is as follows:

Definition 2 (Software Model Validation) *Consider the transition systems $\mathcal{T}_\mathcal{S}$ formed by composing $\mathcal{S}$ and $\mathcal{E}$, and $\mathcal{T}_\mathcal{M}$ formed by composing $\mathcal{M}$ and $\mathcal{E}$. Determine whether $\mathcal{T}_\mathcal{M}$ satisfies Φ only if $\mathcal{T}_\mathcal{S}$ satisfies Φ.*

This work takes a particular approach towards solving this problem, which can be formalized using the notions of simulation and stuttering simulation. Specifically, we use the following result, a proof of which one may find in any standard book on model checking, such as the one by Clarke et al. [123].

Proposition 1 *Given two transition systems $\mathcal{T}_1$ and $\mathcal{T}_2$, and a property Φ of the form $\boldsymbol{G}\phi$. If, using a labeling function $\mathcal{L}$ based on variables appearing in ϕ, $\mathcal{T}_2 \preceq \mathcal{T}_1$, then $\mathcal{T}_2$ satisfies Φ if $\mathcal{T}_1$ satisfies Φ.*

The above result applies also when one uses stuttering simulation instead of "plain" simulation.

Let $\mathcal{V}^* \subseteq \mathcal{V}_\mathcal{S} \cup \mathcal{I}_\mathcal{S}$ be a set of variables deemed to be relevant to proving or disproving that $\mathcal{S}$ satisfies Φ. For soundness, $\mathcal{V}^*$ must contain all variables that influence the value of ϕ. However, this set of variables is difficult to determine exactly. Instead, we generate $\mathcal{V}^*$ based on the "relevant" code fragments left after pruning.

At a minimum, we include in $\mathcal{V}^*$ the set $\mathcal{V}_\phi$ of variables that syntactically appear in ϕ. Conservatively, the rest of $\mathcal{V}^*$ could be computed syntactically from the cone of influence of $\mathcal{V}_\phi$ on the entire code base [123]. However, this can lead to a highly overapproximate set $\mathcal{V}^*$. An alternative is to compute $\mathcal{V}^*$ as the syntactic cone of influence of $\mathcal{V}_\phi$ only on the code *excluding* f_{misc}. In this latter case, if we additionally verify that $\mathcal{V}^*$ is not modified in f_{misc}, we can conclude that there is no code in f_{misc} that influences the value of ϕ. This is the approach we adopt in this work.

stuttering steps in $\mathcal{T}_1$, but we chose this for simplicity and since it fits our problem context.

Label the states of $\mathcal{T}_\mathcal{S}$ and $\mathcal{T}_\mathcal{M}$ using valuations to the variables in $\mathcal{V}^*$. Then, the approach of this work can be outlined as follows:

1. Transform $\mathcal{S}$ to $\mathcal{A}$ by identifying code fragments $\mathcal{F}_\mathcal{P}$ in $\mathcal{S}$;
2. From $\mathcal{A}$, obtain a new program $\mathcal{P}$ by dropping f_{misc} from the set $\mathcal{F}_\mathcal{A}$ and treating any invocation of f_{misc} from f_{orc} as a no-op (stuttering step);
3. Check that $\mathcal{S} \preceq_{\text{st}} \mathcal{P}$ using the labeling function based on $\mathcal{V}^*$;
4. Create a model $\mathcal{M}$ of $\mathcal{P}$ (manually or automatically), where each operation op_i has a 1-1 correspondence with a code fragment f_i in $\mathcal{P}$, and
5. Validate $\mathcal{M}$ by checking that: (i) $\mathcal{I}_\mathcal{M} = \mathcal{I}_\mathcal{P}$; (ii) $\mathcal{V}_\mathcal{M} \supseteq \mathcal{V}_\mathcal{P}$; (iii) $Init_\mathcal{M}$ is equivalent to $Init_\mathcal{P}$ when projected on the common variables, and (iv) $\mathcal{P} \preceq \mathcal{M}$ using the labeling function based on $\mathcal{V}^*$.

The third step above is what we implement as data-centric model validation (DMV), and the fifth step is operation-centric model validation (OMV). Note that we allow $\mathcal{M}$ to have more variables than $\mathcal{P}$ and $\mathcal{S}$, since models often have extra specification variables for use in the proof. If the two simulation checking steps pass, then we can conclude that $\mathcal{M}$ stutter simulates $\mathcal{S}$ and therefore satisfies Φ only if $\mathcal{S}$ satisfies Φ. Note that this is only true for safety properties, which is what we consider in this paper. It is possible for a model that stutter simulates $\mathcal{S}$ to satisfy a liveness property, even though $\mathcal{S}$ does not.

The two simulation checking steps are the most important ones. We implement them as follows:

- Checking if $\mathcal{S} \preceq_{\text{st}} \mathcal{P}$: To do this, it is sufficient to verify that f_{misc} does not modify the values of variables in $\mathcal{V}^*$. There are several ways to do this, and we take the approach of using an assertion checking tool based on symbolic execution, described further in Section 4.3.
- Checking if $\mathcal{P} \preceq \mathcal{M}$: For this, we check that the transition relation $\delta_i^\mathcal{M}$ of each op_i overapproximates that of the corresponding code fragment f_i, denoted $\delta_i^\mathcal{P}$. The transition relations are computed using symbolic execution, and the check $\delta_i^\mathcal{P} \implies \delta_i^\mathcal{M}$ is discharged using SMT solving.

In theory, our methodology is sound, meaning that we never wrongly conclude that a model $\mathcal{M}$ satisfies Φ when the system $\mathcal{S}$ does not. However, in practice, our implementation has encountered significant practical limitations of symbolic execution tools, especially in the DMV step. Therefore, the implementation described in the following sections is used as more of a bug-finding tool rather than a verifier. Nevertheless, we demonstrate that it is useful in practice in weeding out various kinds of bugs in our models.

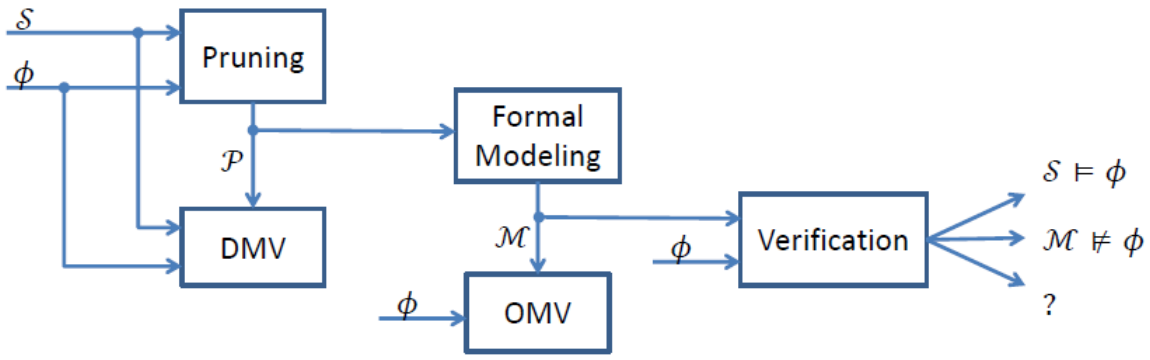

Figure 4.3: The five steps in our model validation process.

Example

Consider applying our problem definition to validate the model in Figure 4.2 against its corresponding program. Assume that we have already performed the DMV step and determined that `update_access_dirty` does not modify any relevant state variables.

Our OMV methodology then produces the following two SMT queries, a query for each pair $(pc, \mathcal{V}_{\mathcal{P}}')$ in the program's symbolic summary. The model is valid if and only if both queries are proven correct by the backend solver.

1. $(page\_fault' = ITE(cpl[0] = 1 \wedge cpl[1] = 1, 1, 0) \wedge cpl = 3) \Rightarrow (page\_fault' = 1)$
2. $(page\_fault' = ITE(cpl[0] = 1 \wedge cpl[1] = 1, 1, 0) \wedge cpl \neq 3) \Rightarrow (page\_fault' = 0)$

The first query is proven valid by our SMT solver. However, the second query is invalid, indicating that our model does not correctly capture the behavior of the code. The returned counter-example is this: when $cpl = 7$, the two low-order bits are both set, and therefore, *page_fault* is set. The model makes an implicit assumption that *cpl* will never be higher than 6. This seems reasonable, considering a physical CPU's CPL is never higher than 3. However, the C code enforces no such assumption, and so the model is incorrect.

4.3 Implementation

In Section 4.2, we outlined our formal approach to validating a model. In this section, we describe our practical implementation. Figure 4.3 depicts the data flow of the process.

Pruning the System

The pruning step of our implementation corresponds to steps 1 and 2 of the outline given in Section 4.2. In our implementation, the pruning stage is manual, and performed by a domain expert. Given the system, $\mathcal{S}$, and the property, Φ, the expert decides which code fragments are relevant and creates $\mathcal{A}$, with the set $\mathcal{F}_{\mathcal{P}}$ identified. The expert then returns $\mathcal{P}$, in which code in f_{misc} has been replaced with no-ops.

DMV

Although pruning is performed by people with expert knowledge of $\mathcal{S}$, the process is error prone. Some function or module in $\mathcal{S}$ that is relevant to Φ may get overlooked and be mislabeled as f_{misc} during the annotation of $\mathcal{A}$. DMV takes as inputs $\mathcal{S}$, Φ, and $\mathcal{P}$ and returns a Yes/No answer, indicating whether $\mathcal{P}$ and $\mathcal{S}$ are equivalent with respect to Φ. If the answer is No, the domain expert repeats the pruning stage until DMV can return Yes. The step corresponds to step 3 of the theoretical approach given in Section 4.2. In that step we prove $\mathcal{S} \preceq_{\text{st}} \mathcal{P}$ by checking that f_{misc} does not modify the values of variables in $\mathcal{V}^*$. In our implementation, we relax this requirement, and check only for a subset of $\mathcal{V}^*$. As a result, we can not prove the simulation, only find instances when it fails.

We start by identifying the program variables $\mathcal{V}_\phi$ that the property Φ directly depends on. This can be done syntactically: the variables in $\mathcal{V}_\phi$ are those variables mentioned in ϕ. We wish to establish that any state change described in f_{misc} does not affect, directly or indirectly, the state described by $\mathcal{V}_\phi$; i.e., those pruned state changes are *stuttering steps* with respect to $\mathcal{V}_\phi$. To do this, we attempt to show variables in $\mathcal{V}_\phi$ do not change value within f_{misc}. We tackle this by keeping track of the last valuation of $\mathcal{V}_\phi$ and ensuring that it does not change after any statement that appears in f_{misc}.

More specifically, we create a system $\mathcal{S}'$ that is identical to $\mathcal{S}$, except for the following modifications. For each variable $v \in \mathcal{V}_\phi$, we create a new shadow variable v' in $\mathcal{S}'$. After every (atomic) statement in $\mathcal{S}'$ that is also in $\mathcal{P}$, we add a statement that assigns the value of v to v'. In other words, we attempt to keep the value of v' synchronized with the value of v across $\mathcal{P}$ statements in $\mathcal{S}'$. In contrast, after every atomic statement in $\mathcal{S}'$ that is a statement in f_{misc}, we add an assertion that v equals v' for each $v \in \mathcal{V}_\phi$, which asserts that these statements do not write to any variable in $\mathcal{V}_\phi$, or equivalently, that every statement in f_{misc} is a stuttering step with respect to $\mathcal{V}_\phi$. Note that, since v' variables do not appear in $\mathcal{S}$ and do not alter the control flow, they do not alter the behavior of $\mathcal{S}'$ with respect to $\mathcal{S}$ state.

We then use KLEE to search for a path through $\mathcal{S}'$ where one of the inserted assertions fails. If KLEE is successful, we have found a f_{misc} statement that updates $\mathcal{V}_\phi$, so our DMV algorithm returns No. The path and inputs that cause the violation are passed back to the pruning stage, allowing the expert to refine the definition of $\mathcal{P}$ to incorporate the previously

missed $\mathcal{V}_\phi$ update.

In practice, the extra statements in $\mathcal{S}'$ increase the size of the original system, $\mathcal{S}$, by $|\mathcal{V}_\phi|$, which complicates and prolongs the execution time of the dynamic analysis performed by KLEE or other symbolic-execution engines. What is more, many of the added statements are ineffectual: in most practical systems, $\mathcal{V}_\phi$ will not be updated in every single statement, so checking even one assertion per statement can be overkill. To ease the burden on the symbolic execution engine, our prototype relaxes the strict definition of $\mathcal{S}'$ and aims to capture commonly missed $\mathcal{V}_\phi$ updates, but at the expense of soundness. Specifically, our prototype adds $v' := v$ statements into $\mathcal{S}'$ only after $\mathcal{P}$ statements that the domain expert knows to be updating v. Furthermore, our prototype adds $v' = v$ assertions into $\mathcal{S}'$ only before $\mathcal{P}$ statements known to be updating v (contrast that to the original definition, which adds assertions after every f_{misc} statement). The rationale for this last relaxation is that updates to v' are good enforcement points for the equality between v' and v, and this requires fewer changes to the program. Finally, our prototype checks the $v' = v$ assertion at all exit points from $\mathcal{S}'$, to capture any $\mathcal{V}_\phi$ updates on execution suffixes that do not include a known $\mathcal{P}$ update.

Our prototype's relaxation is unsound. It may miss successive updates to a v between known updates to it; if a statement reads such a missed v update, affecting the result of $\mathcal{P}$, our validation down the pipeline may be unsound as well. Since our prototype is a bug-finding tool inspired by our idealized methodology, this unsoundness is acceptable for our purposes. A more effective, sound relaxation would be to assert that $v = v'$ before every *read* of v, but finding all reads of v is, in itself, another hard problem, with similarities to alias analysis. The use of dynamic slicing [124] may also be useful.

Modeling the Program

From the selected $\mathcal{P}$, a model $\mathcal{M}$ is built using UCLID's modeling language. We are agnostic as to how the modeling is done; it can be a manual or an automated process. This corresponds to step 4 of the outline given in Section 4.2.

Validating the Model

In the last step – step 5 of the outline in Section 4.2 – we validate that the model $\mathcal{M}$ correctly simulates the program $\mathcal{P}$. We use KLEE for this step. KLEE is a symbolic interpreter for LLVM [125] programs. It performs a combination of symbolic and concrete execution, based on which inputs and state bits are made symbolic. We use KLEE to learn $R = \{(pc_{\mathcal{P}}, \mathcal{V}_{\mathcal{P}}')\}$, the set of path conditions and corresponding outputs describing the input–output relation of $\mathcal{P}$. First, we set all global state $\mathcal{V}_{\mathcal{P}}$ and input variables $\mathcal{I}_{\mathcal{P}}$ to be unconstrained, symbolic values. When execution reaches a conditional branch point, KLEE forks and follows all feasible branches. As KLEE explores these program paths, it maintains a path condition for

each path that is a function of $\mathcal{V}_\mathcal{P} \cup \mathcal{I}_\mathcal{P}$. When a path terminates via an exit statement or at the end of the program, we note the path condition $pc_\mathcal{P}$ for that path and the state of $\mathcal{V}_\mathcal{P}'$ at the point of termination. Once KLEE has explored all possible paths, we have R.

Once we have computed R, we use UCLID's decision procedure to verify that for every path condition describing a partition of the input space in $\mathcal{P}$, both $\mathcal{M}$ and $\mathcal{P}$ produce equivalent symbolic output. Our queries to the UCLID decision procedure take the form of $\texttt{decide}(pc_\mathcal{P} \Rightarrow (\mathcal{V}_\mathcal{M}' = \mathcal{V}_\mathcal{P}'))$, and we have a separate query for each $(pc_\mathcal{P}, \mathcal{V}_\mathcal{P}')$ pair in R. If one of the queries fails, UCLID will return a counter-example with concrete values for $\mathcal{I}_\mathcal{P}$ and $\mathcal{V}_\mathcal{P}$ that satisfy the path condition, but for which $\mathcal{V}_\mathcal{M}' \neq \mathcal{V}_\mathcal{P}'$.

Translating Path Conditions to SMT queries

In order to create each query, we must first translate the $pc_\mathcal{P}$ generated by KLEE to a format suitable for UCLID's input language. KLEE's path conditions are written in KQuery,[4] the input language to KLEE's backend constraint solver (Kleaver). The symbolic states of $\mathcal{V}_\mathcal{P}$ are given in a similar syntax. We built a tool to translate path conditions and symbolic state from KLEE into SMT queries in UCLID's input language. Our tool requires the user provide an input file that maps variable names used in $\mathcal{P}$ to variable names used in $\mathcal{M}$. All variables in KQuery are represented as an array of bits; these are easily translated to UCLID's bit-vector type. UCLID allows two additional types: Boolean, and uninterpreted functions. Currently, we require UCLID models to not use Boolean types; bit-vectors of length 1 can be used instead. Seamlessly converting between the two is functionality that can be added to future versions of our tool. UCLID models are free to use uninterpreted functions, but we do not handle verification with respect to properties that include universal quantification over an uninterpreted function. For example, if f is an uninterpreted function in a UCLID model and i is a bit-vector variable in that model, our tool can validate a model with respect to the property $\phi(f(i))$, but would not be able to validate a model with respect to the property $\phi(f)$.

Handling Loops

For programs that contain loops, the number of paths possible through the code can explode quickly. In cases where the loop bound can be determined statically, this is not a problem. However, when the loop bound is determined at run time, complete path coverage requires exploring all possible loop bounds, causing an exponential blow-up in the number of paths through the program.

In some cases we can handle this using a divide & conquer approach. Figure 4.4 illustrates the idea. For a program $\mathcal{P}$ with a single loop, we can divide the source code of the program into three parts: the code from program entry to the loop, the loop code itself, and the code

[4] http://klee.llvm.org/KQuery.html

```
1 int loop_prog(int bound) {
2    int retval = 0;          //E2L, lines 2,3,4
3    if (bound <= 0)
4       return -1;
5    for (int i = 0; i < bound; i++)
6       retval++;             //L2L, lines 5,6
7    return retval;           //L2E
8 }
```

Figure 4.4: A program with a dynamically determined loop bound. Validation of this program's model is done in three parts.

from loop exit to program exit. In Figure 4.4, these sections are labeled E2L, L2L, and L2E, respectively. We then explore each section of code independently of the other two. At the start of each exploration we set the current constraints on $\mathcal{I}_{\mathcal{P}}$ and $\mathcal{V}_{\mathcal{P}}$ to reflect the invariants of the previous section(s). These invariants are determined manually and care must be taken to not over-constrain the starting conditions of each section. At a minimum, the following weak invariants can safely be used: no constraints set for E2L, the loop guard forms the constraints for L2L, and no constraints set for L2E. These constraints will form part of the path conditions created during subsequent exploration of the section. For example, in Figure 4.4, execution of E2L starts with no constraints on *bound* or *retval* and the starting path condition is $pc_{\text{E2L}} = \texttt{True}$; execution of L2L is started with the path condition (and corresponding constraints) $pc_{\text{L2L}} = (i \geq 0) \wedge (i < bound)$; and execution of L2E is started with no constraints.

We divide the model into three parts as well. We create a separate model for the E2L, L2L, and L2E program fragments, and separately validate each model against its corresponding program fragment. Because each fragment is explored starting from an overapproximation of reachable starting states, the explored paths through the fragments will constitute a superset of reachable paths in the original program. If the model is shown to be consistent with each of these paths plus the corresponding output, a property proven true of the model will be true of the program. However, this method is not complete, and a property that is disproven for the model may in fact be true of the program.

In some cases, it may be possible to strengthen the invariant through manual inspection of the code. For example, in Figure 4.4, the execution of L2L can be started with the path condition $pc_{\text{L2L}} = (i \geq 0) \wedge (i < bound) \wedge (bound > 0) \wedge (retval \geq 0)$, and execution of E2L can be started with the path condition $pc_{\text{L2E}} = (i \geq bound) \wedge (bound > 0) \wedge (retval \geq 0)$. This will increase the completeness of model validation for some programs, but still does not guarantee it. Furthermore, our divide & conquer approach is suitable to code with single loops, but is not applicable to code with nested loops or recursion.

4.4 Evaluation: Data-Centric Validation

BPF

The Berkeley Packet Filter (BPF) is a kernel module that filters network packets, sending only the desired ones to the user. The user provides a filter program, written in the BPF pseudo-machine language. The BPF kernel module contains an interpreter that runs the filter program on each network packet. The property Φ asserts that the BPF interpreter, *bpf_filter* (Figure 4.6), has a monotonically increasing program counter; this is desirable, since it implies that all filter programs must eventually halt. We restricted our analysis to the kernel mode version of the code, since the user mode version is known to violate this property (e.g., for efficiently implementing IPv6 protocol header chain parsing), though we did not enable "zero-copy memory buffers," which we considered out of scope.

One challenge is that it is not practical to explore all reachable states by running the BPF interpreter on all filter programs of n fully symbolic instructions. Therefore, we overapproximate its behavior: we run one iteration of the interpreter loop on a single, fully symbolic BPF instruction, starting from a fully symbolic state. This overapproximation may lead to spurious counter-examples involving states that are not ordinarily reachable.

In particular, we made the accumulator, the index registers, the temporary variable k, and the memory fully symbolic. The inputs were fully symbolic, two-instruction BPF programs that had passed the *bpf_validate* function (Figure 4.5).[5] We encountered no spurious counter-examples. We were able to efficiently explore all paths of *bpf_filter* using KLEE in less than 5 seconds and confirm that we had identified all writes to the program counter. Thus, our data-centric model validation was successful.

ftpd

ftpd is GNU's File Transfer Protocol server, included in their Inetutils package (v1.8). The main loop reads input from the network client, parses the command, and runs the appropriate block of code to respond to the client. The called code may be inline or in a separate function.

It would be interesting to model the blocks of code that affect whether the user is considered to be logged in. There are many FTP commands, but we would expect few to be relevant – for example, *PASS* (authentication password), but not *PWD* (print working directory) – thus being able to safely prune functions would greatly reduce the amount of modeling.

A simple syntactic approach of searching for "*logged_in* =" identifies only the *PASS* function (Figure 4.7), so we used our DMV approach to check for other writes to *logged_in*.

[5] A valid BPF program must end with a *return* instruction, so our input approximates a fully symbolic "one-instruction" program.

```
1  int bpf_validate(filter, len) {
2      for (i = 0; i < len; ++i) {
3          switch (BPF_CLASS(filter[i].code)) {
4              case BPF_ST:
5                  if (filter[i]->k >= BPF_MEMWORDS)
6                      return 0;
7                  break;
8              case BPF_DIV:
9                   // Check for division by 0.
10                 if (filter[i]->k == 0)
11                     return 0;
12                 break;
13             ...
14         }
15     }
16     return BPF_CLASS(filter[len-1].code) == BPF_RET;
17 }
```

Figure 4.5: Simplified code from *bpf_validate*, which checks that the BPF program cannot write out-of-bounds, cannot divide by zero, and ends with a *return*.

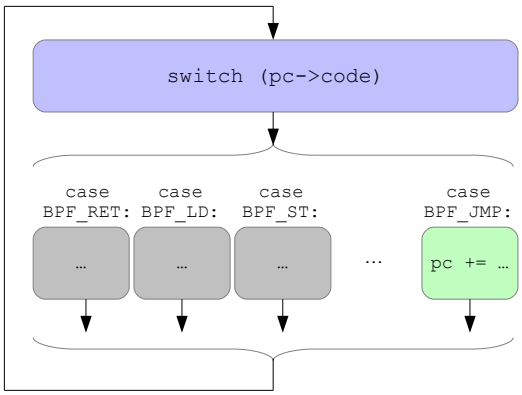

Figure 4.6: The *bpf_filter* function interprets instructions one at a time. We used DMV to confirm that only the *BPF_JMP* family of instructions modifies the program counter.

```
pass (const char *passwd) {
    if (cred.logged_in || askpasswd == 0)
        return;
    askpasswd = 0;
    cred.logged_in = 1; }

user (const char *name) {
    if (cred.logged_in)
        end_login (&cred);
    askpasswd = 1; }

end_login (struct credentials *pcred) {
    memset (pcred, 0, sizeof (*pcred)); }
```

Figure 4.7: Simplified pseudo-code for *pass* and *user* functions in the *ftpd* software.

We modified the *ftpd* software so that its input was partly symbolic: it could choose to explore any of *USER* and *PASS* (with a prespecified, known-good combination), nine other commands that had no parameters (e.g., *NOOP*, *PWD*, *QUIT*), and nine commands with partly symbolic parameters (two alphanumeric characters). We excluded other commands because there were parsing issues with some parameters, they were not implemented by *ftpd*, or they modified the system (e.g., *RMD*), which is not well handled with KLEE. In some cases, we also modified functions (e.g., *CWD*) to remove system calls or side effects. We then added the shadow credentials data structure,[6] copying *cred* to *shadow_cred* after the known write to *logged_in*, and placed "*assert (cred.logged_in == shadow_cred.logged_in);*" before the write and in the body of the parser loop.

We started the analysis from a partly symbolic state, with the *askpasswd* global variable and all of the credentials data structure made symbolic (for each string, we stored two symbolic characters). KLEE produced a test case where the assertion failed: when the client is already logged in, and issues another *USER* command, the entire credentials data structure is zeroed out by END_LOGIN() (Figure 4.7). This is non-trivial to identify syntactically, because unlike the previous case where *cred.logged_in = 1;*, this write syntactically involves neither *cred*, *logged_in*, nor =.

After adding the update to *shadow_cred.logged_in* after the write in the *user* function, we verified with DMV that we had not missed any writes. In a separate experiment, we confirmed that DMV would also have been able to identify a missing write to *cred.logged_in* in the *PASS* function. Each experiment required less than 3 minutes of run-time. Data-centric

[6]It is tempting to place the shadow copy of *logged_in* inside the original credentials (i.e., *cred.shadow_logged_in*), but as will be observed, this would be unwise as any writes to *cred* may also overwrite the shadow variable.

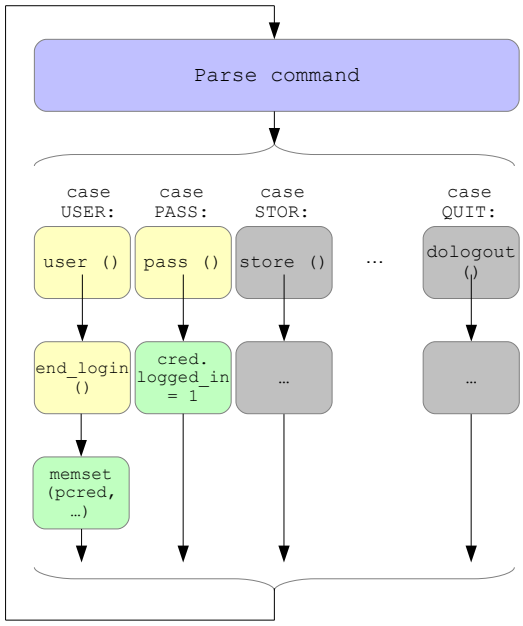

Figure 4.8: *ftpd* interprets commands one at a time. Only the *USER* and *PASS* commands modify the *logged_in* variable. This is significantly more complex than the BPF example, with some cases resulting in nested function calls across multiple files.

model validation therefore has been able to identify a missed write to *cred.logged_in*, and provide some degree of confidence that none of the blocks of code that were explored, including those corresponding to FTP commands, the parser implementation, and some dependencies, do not write to this variable (Figure 4.8).

XMHF hypervisor

The eXtensible Modular Hypervisor Framework (XMHF) [93] is a hypervisor with a small trusted core of about 6000 lines of code that is amenable to formal verification. To demonstrate this, Vasudevan et al. [93] have proven a memory integrity property of XMHF. A hypervisor runs at a higher privilege level, and supports the execution of guest software running at a lower privilege level. Whenever a guest executes a privileged instruction (e.g., to set the CR3 register), the CPU transfers control to the hypervisor, which uses an intercept handler to take the necessary action. The XMHF intercept handler can be summarized by Figure 4.9. The intercept handlers are worth modeling to verify that their updates do not violate a security property.

Consider the following property: the extended page tables must remain constant after setup. This property requires us to model all program paths in *interceptHandler* that

```
void interceptHandler(VCPU *vcpu,struct registers* regs){
  switch(vcpu->vmcs.vmexit_reason) {
    case CRX_ACCESS: handle_crx_access(vcpu,regs); break;
    case IO: handle_ioport_access(vcpu,regs); break;
    case WRMSR: handle_wrmsr(vcpu,regs); break;
    case EPT_VIOLATION: handle_pagetable(vcpu,regs);break;
    ... }}
```

Figure 4.9: Simplified code for XMHF intercept handler.

might update relevant state such as the *vcpu→vmx_vaddr_ept_pml4_table* field, which shadows the address pointing to the base of the extended page tables. Instead of modeling all intercept handlers, we like to prune away handlers that provably do not update the *vcpu→vmx_vaddr_ept_pml4_table* field. Forcing the input data structures *vcpu* and *regs* to be symbolic, we use KLEE to symbolically explore paths in *interceptHandler*. DMV found no writes to the *vcpu→vmx_vaddr_ept_pml4_table* field in each handler that the expert decided to prune away. This allows us to prune away the following handlers: set CR0 register, set CR4 register, read MSR register, write MSR register, read CPU id, access IO ports. This experiment used KLEE to evaluate 300 lines of C code in 5 seconds.

Note that the intercept handlers rely on inlined assembly instructions to perform certain operations such as reading the MSR register. Since we do not model the x86 hardware, we assume that none of the inlined assembly operations impact the *vcpu→vmx_vaddr_ept_pml4_table* field. While we are confident of this assumption, we do recognize this as a potential weakness of this case study.

4.5 Evaluation: Operation-Centric Validation

Bochs address translation function

Bochs [54] is the open-source x86 emulator whose address translation function we verified in Section 3.4. In this case study, we validate that the model we used in Chapter 3 correctly overapproximates Bochs' address translation function.

The address translation function in Bochs is 98 lines of code and the corresponding UCLID model is about 300 lines of code. The set of inputs contains three elements: the linear address to be translated, read or write permissions requested of the address, and the emulated CPU's current privilege level. The Bochs linear to physical address translation requires either a two-step page table walk or a lookup in the translation lookaside buffer (TLB), a cache of previously translated addresses. We make symbolic the three inputs, the two page table

entries, and the single TLB entry that the translation could access, plus all global variables. Because our model validation is motivated by our previous verification effort, we use the returned physical address as the output of the function, and our model validation checks whether the physical address returned by the model is always equivalent to the physical address returned by the code.

During symbolic execution, KLEE explored 219 paths through the code.[7] KLEE achieves full code coverage of the function, as seen in Table 4.2. There is one branch not taken by KLEE; manual analysis reveals that taking this branch would not lead to any new state in the code. For each of the 219 paths explored by KLEE, we captured the path constraints describing the path and check whether the symbolic state of the physical address at the end of execution of the C++ function is equivalent to the physical address resulting.

Model validation found seven discrepancies between the behavior of the source code and the behavior of the model, each traced to a model bug. We categorize these bugs by the likely cause of the modeling error; there was one typo, two implicit assumptions, and four logic errors. In the assumptions, the modeler made some implicit assumption about either an invariant of the system or a pre-condition to the function being modeled. One of these was the bug we used in our running example (Section 4.1) in which the modeler assumed the CPU privilege level was never greater than 3. While this is a reasonable assumption for x86, it causes the model and code to have divergent behaviors. In one logic error, an incorrect ordering of case statements in the model caused incorrect behavior. Table 4.3 summarizes the results of the model validation, while Table 4.1 shows the source and model code for six of the seven bugs. The seventh bug, not shown here, was an error in a large case statement.

Code	Model
Typos	

`if (!(pte & 0x1)) {`
`  CPU->pagefault = 1;`
`  return 0;`
`}`
`...`
`if (!(pde & 0x1)) {`
`  CPU->pagefault = 1;`
`  return 0;`
`}`
	`next[pagefault] := (priv_check = zero_bit) |`
	`((pde # [0:0]) = zero_bit) |`
	`((pde # [0:0]) = zero_bit);`
	
	`/* Model reads bit 0 of pde twice`
	`and never reads bit 0 of pte.`
	
	`The syntax 'pde # [0:0]' extracts`
	`bit 0 from the pde bitvector. */`

False Assumptions	

[7] Ignoring inline functions that gcov counts as branches.

```
unsigned pl = (curr_pl == 3);
```

```
pl := (curr_pl # [0:0]) &&
      (curr_pl # [1:1]) );

/* Model assumes curr_pl is less than 7.

The syntax '&&' denotes a bit-wise
AND operation. */
```

```
bx_bool isExecute = (rw == 2);
```

```
isExecute := (rw # [1:1]) &&
             !!(rw # [0:0]);

/* Model assumes rw is less than 6.

The syntax '!!(rw ...)' denotes
bit-wise negation. */
```

Logic Errors

```
combined_access = (pde & pte) & 0x06;
...
priv_index = /* value of priv_index
              depends on combined_access */

if (!priv_check[priv_index]) {
    CPU->pagefault = 1;
    return 0;
}
...
combined_access |= (pte & 0x100);
```

```
combined_access := (
  ((pde && pte) && 0x06)
       || (pte && (0x100)) );
...
priv_check := /* value of priv_check depends
              on combined_access */

next[pagefault] :=
 (priv_check = zero_bit) |
 ((pde # [0:0]) = zero_bit) |
 ((pte # [0:0]) = zero_bit);

/* Model updates combined_access with all
writes before checking for pagefaults. */
```

```
if (/* TLB hit and permissions OK */)
  return paddress;
else
  /* page walk */
...

/* Check for pagefault conditions
that occurred during page walk. */
if (!(pde & 0x1)) {
  CPU->pagefault = 1;
  return 0;
}
if (!(pte & 0x1)) {
  CPU->pagefault = 1;
  return 0;
}
if (!priv_check[priv_index]) {
  CPU->pagefault = 1;
  return 0;
}
...
return paddress;
```

```
init[paddress] := 0x0;
next[paddress] :=
case
  next[pagefault] : 0x0;
  ~permission|~present : /* page walk */
  default : /* TLB entry */
esac;

/* The model checks whether a page walk
would incur a pagefault without first
checking if there is a TLB hit, which
would make the page walk unnecessary.
The corrected model puts the TLB check
first, then the check for pagefaults,
and finally, the default action is to
return the address calculated by a page
walk. */
```

```
if (! (tlbEntry->accessBits &
       ((isExecute<<2) |
        (isWrite<<1) | pl)) ) {
  /* TLB entry has correct
  permissions. */
}
```

```
permission :=
   (!! ((tlbentry(TLB_index) # [98:96]) &&
        ((isExecute << 2) || (isWrite << 1)
         || pl))) != 000;

/* Model uses a complicated double
negative that evaluates incorrectly. */
```

Table 4.1: Bochs modeling bugs (6 of 7)

TCAS

Traffic Collision Avoidance Software (TCAS) [126, 127] is an aircraft collision avoidance system that is in use onboard all large aircraft today. Its use is complementary to air traffic control and is meant to reduce the occurrence of mid-air collisions. A TCAS system consists of an onboard transponder, which is used to detect and communicate with other aircraft, and the TCAS software, which calculates the necessary corrective action required by the pilot in the case of impending collision with another aircraft. The TCAS software is complex and safety-critical, and verification of its correctness is well-suited to model checking. In this case study, we used a simplified and publicly available version of TCAS that the software engineering research community uses to evaluate the efficacy of various test case generation

techniques [128]. Though only 133 lines of C code,[8] the code includes 9 functions and has some complicated patterns of control flow.

From the TCAS code, we created 23 different models, each injected with a different fault developed by Hutchins et al. [129]. The 23 models were chosen as a subset of the 41 faults in Hutchins test suite. We chose the subset to avoid duplication of similar faults. For example, if Hutchins' set contained two related but different faults, such as replacing $x = 300$ with $x = 400$ or with $x = 500$, we used only one of those faults. Each fault is meant to mimic the type of bug a programmer might accidentally introduce while building the model. We use these 23 different versions to evaluate the effectiveness of model validation at finding these programmer errors; we validate each model against the TCAS source code, and check whether model validation was able to find the injected fault. To avoid leading our model validation technique to find the known bug, we had one author create all the models and a second author complete the model validation.

We made symbolic all 12 inputs to the TCAS program. During symbolic execution, KLEE explored 29 paths through the code. There was one line of code not covered by KLEE, see Table 4.2, and manual analysis revealed this line of code to be unreachable. It is an error state in which the suggested corrective action for the pilot is to both climb and descend. There were eight branches not taken by KLEE and these were either infeasible or redundant branches. Redundancy in branches is an artifact of how gcov counts branches. For example, the statement `if (A && B)` has two possible branches: taken and not taken. To cover the not taken branch, it is sufficient to find a path in which either `A` or `B` is false. However, each logical conjunction or disjunction is counted as introducing a new branch in gcov, and if KLEE does not explore all possible combinations of `A` and `B`, gcov considers those unexplored combinations to be branches not taken, even though all actual branches through code may have been explored.

Our model validation found faulty behavior in all 23 models. It is difficult to categorize these bugs since they did not naturally arise during the translation from code to model. However, they included: using a less-than-or-equal ($\leq$) sign in place of a less-than sign ($<$), or vice versa; using a logical AND (&) in place of a logical OR (|), or vice versa; leaving an entire clause out of a series of conjunctions and disjunctions; and typographical errors such as setting a variable to the wrong value. In addition to the 23 injected faults, model validation also revealed two model bugs that were introduced unintentionally during the modeling phase, see Table 4.3. That is, neither the modeler nor the validator were expecting these bugs to be in the model. The first bug was a typographical error: a variable that should have been set to 0x2e4 was instead set to 0x284. The second bug was a logic error. An `int` variable in the code was used as a flag: non-zero is true and zero is false. In the model, greater than zero was treated as true, while zero or less was treated as false.

[8] Calculated using SLOCCount (http://www.dwheeler.com/sloccount/).

BPF

The Berkeley Packet Filter (BPF) uses an interpreted filter program to filter network packets. The interpreted pseudo-machine language includes instructions for load/store, ALU operations, branch, and return. A badly or maliciously written program could crash the BPF interpreter. Prior to running a filter, BPF runs a validator to ensure the filter is valid and satisfies three properties: 1) any jumps in the program move forward (no loops are allowed), 2) any jumps move to a legitimate place in the program (i.e., do not jump past the end the program), and 3) the program always terminates with either an accept or reject. Validate before use is a common strategy for security-critical code. For example, software fault isolation may compile a program to obey strict control flow policies and then statically validate just the compiled program [130–133]. The benefit of this approach is that it requires trusting only the relatively small validator and not the large compiler. In this scenario, as with BPF, it is important to verify that the validator will catch any failing programs.

In this case study, we show how one might verify that filter programs are marked valid if and only if they satisfy the three properties above. We build a model of the validator for use in a model checking-based proof. Before doing the model checking, though, we want to validate our model to make sure it accurately represents the validator function.

Program validators commonly loop over each program instruction, processing it before examining the next. The loop's bound typically can not be determined statically, making full path exploration using symbolic execution challenging. Non-statically bounded loops are not commonly seen in hardware designs and so are not handled by model validation techniques used for hardware designs. The *bpf-validate* case study demonstrates the validation of this software feature.

bpf-validate takes two inputs, filter program pointer, and filter program length. It has two local variables, current program instruction pointer, and loop iteration counter. Because the for-loop does not have a bound that can be determined statically, we use the divide & conquer technique for model validation outlined in Section 4.3. We transform the single *bpf-validate* function into three separate functions: E2L, L2L, and L2E and we build a model with three separate modules, one for each new function. E2L sanity checks the *length* parameter. L2E checks that the filter program ends correctly with a *return* instruction. L2L does the bulk of the filter program validation. Each iteration of the loop checks a new instruction in the BPF program to make sure it is valid, does not perform illegal arithmetic such as division by zero or illegal memory accesses, and does not jump to an illegal point in the program. We also introduce a new return value *return_happened* to indicate whether the next function should execute. For example, if *return_happened* is 1 after execution of E2L in a particular run, the execution of L2L and L2E are unnecessary; the original composed code would not have gotten that far.

For all these functions, we create a program of length one (a single instruction). We make the contents of that program fully symbolic, and we make the *length* parameter symbolic as

well. We constrain *length* to be less than or equal to the length of the program. Omitting this constraint causes KLEE to find the potential memory error when the for-loop tries to access the next instruction in the program at an array index that is out of bounds. This is useful for checking the memory safety of the C program, but not relevant for our model validation.

Because our technique assumes a bounded input, we bound the length of our BPF program to a single instruction. Coming up with a reasonable size for the bound will depend on the code being modeled; in this case the *bpf-validate* function only accesses a single instruction in each iteration of the loop, and saves no state from the processing of one instruction to the next, so a bound of one is reasonable. At the start of L2L and L2E, we add constraints on the symbolic variables to match the program constraints at that point in the code. For example, if execution makes it to the start of the for-loop, neither of the preceding *if* statements were true, else the program would have exited, and so we constrain the symbolic variables with the negation of the *if* conditions. Similarly, at the start of L2E, we constrain the index of the current instruction to be greater than or equal to the *length* variable.

Model validation found four bugs in the model, all in the for-loop. Two of the bugs involved an inverted logic statement. The third and fourth were errors in calculating the return values, *return* and *return_happened*. Both are computed using switch statements in the model, and in both cases there was a case missing.

Program	Lines of Code	Lines Executed	Branches	Branches Taken
Bochs	98	98	30	29
TCAS	60	59	68	60
BPF-validate	71	69	28	24

Table 4.2: Code and path coverage achieved during symbolic exploration of the code by KLEE. The total lines of code given refers only to the pruned program, not the original system.

Program	Typo	Logic Error	Assumption about System Invariants	Total
Bochs	1	4	2	7
TCAS	1	1	-	2
BPF-validate	-	4	-	4

Table 4.3: Model bug types found during operation-centric model validation. For TCAS, only the bugs introduced unwittingly during the modeling phase are categorized.

4.6 Related Work

Techniques for validating a model against an implementation have been explored in the hardware community. In that setting, the goal is to show equivalence between designs written in a high-level language like C, and register transfer level (RTL) implementations written in a hardware description language, such as Verilog or VHDL. The basic idea is similar to our operation-centric model validation: derive an expression, for both the C program and the RTL program, describing the input–output transition relation of the program and use a SAT solver to show equivalence between the two expressions.

Clarke and Kroening describe how to transform both a C function and a corresponding RTL implementation to a bit-vector expression of the transition relation and use a SAT solver to prove equivalence between the two [134]. Others have shown how to use symbolic simulation to derive an expression describing the set of input–output relations of a C function and show equivalence to the RTL [135–139]. In all these cases, it is assumed that the C function is written for eventual implementation in hardware and therefore some restrictions can be placed on the expressiveness of the C code. For example, the works of Clarke and Kroening [134], and Koelbl et al. [137–139] require that all loops and recursive function calls have bounds that can be determined statically, so that they can be unrolled before deriving the input–output transition relation. Feng [135] and Hu et al. [136] assume the software model and corresponding RTL implementation can be reduced to a combinational equivalence problem.

Model validation in the software community has been less widely studied. Heitmeyer et al. show that partitioning the code, and concentrating verification efforts only on those portions of code that are relevant to the property can greatly reduce the verification effort [75, 76]. This is similar in spirit to our pre-processing phase, although our approach is automated and focused on bug-finding, whereas Heitmeyer et al. provide a proof that only code within the verified partition is relevant for the property at hand. The proof is built manually and relies on the source code being fully annotated with pre- and post-conditions for all functions. Their technique applies well for a smaller code base – they demonstrate their approach on an embedded device kernel that is approximately 3 KLOC of C and assembly combined – and might not scale well to software systems as large as Bochs or XMHF. Static analysis techniques such as alias analysis, Mod/Ref analysis, and program slicing (e.g., [124,140,141]) can be used to perform DMV and scale to large code bases, but at the cost of much lower precision, resulting in a high false alarm rate.

Caballero et al. [142] use a combination of concrete and symbolic exploration of a function at the binary level to derive a set of path predicates that describe the behavior of that function. They focus on security-sensitive functions, and in particular filtering functions, and use the derived set of path predicates of one function and a manually written model of a second function to examine whether the two functions implement the same filter. Their set of path predicates is not complete, and rather than proving equivalence between the two versions,

they are focused on finding violations of equivalence (i.e., bug finding). They use a decision procedure to find any instances that will be blocked by one filter, but allowed through by the other.

Outside of model validation, using symbolic execution to compare two versions of the same software code block has applications for checking code optimizations, version tracking, regression testing, and code refactorization. In these settings, there tend to be significant similarities between the two versions being compared, which can be exploited during the validation process; such similarities cannot be relied upon in our model validation setting. Ramos and Engler extend KLEE to run on two different blocks of C code, saving the symbolic state of each run for comparison [143]. This is similar in spirit to our operation-centric model validation technique, although we are comparing C code to a UCLID model, rather than comparing two blocks of C code.

Siegel et al. showed how to use symbolic execution with the SPIN model checker to verify a parallel implementation of numeric computations was equivalent to the original sequential implementation [144, 145]. They impose a bound on the number of iterations taken by any loop and further reduce the complexity of the verification by only accepting as equivalent two symbolic expressions that can be made the same through simple symbolic algebra. In cases where more sophisticated theorem proving techniques are required to prove equivalence, their tool may return "not equivalent."

Person et al. show how to use symbolic execution to characterize differences between two versions of the same software [146]. Their definition of functional equivalence is analogous to our equivalence defined for operation-centric model validation. They reduce complexity of the symbolic exploration by using uninterpreted functions to describe blocks of code that are textually identical in the two versions being compared. Since we are not comparing two versions of the same code, but rather code and a newly generated model, using this technique of overapproximating the software's behavior does not work for us. Instead, we use our divide & conquer approach for handling data-path complexity such as loops whose bound can not be determined statically.

4.7 Conclusion

We have proposed a formal framework for validating a model of a software system against the code it was created from. Using an implementation based on symbolic execution and SMT solving, we have demonstrated the utility of our approach on several case studies. Among the many directions for future work, we believe the most important is to improve the implementation of the DMV check, potentially by combining scalable static analysis methods with selective assertion checking.

Chapter 5

Conclusion

Virtualization software provides isolation properties that make the virtualized environment well suited to a wide variety of security applications. In this work, we tackle the problem of verifying those isolation guarantees. We focus on memory isolation and show how a combination of abstraction and model checking can be used to verify the correctness of memory management modules. This is software that typically involves very large data structures, and that has not been amenable to previous model checking-based techniques. One weakness of a model checking-based approach is that the verification is completed for the model, not the original source code. We address this issue by presenting a framework and implementation for validating the model against the source code. We show that a property proven true of the validated model will also be true of the original source code.

Although motivated by virtualization software, our work has applications beyond CPU emulators and VMMs. Our small and short world verification technique is potentially applicable to any system involving large, symmetric data structures. And, model validation is useful any time a model-based verification technique is used. The results of our case studies suggest that even models of small, well understood programs are likely to have some errors; we propose that adding model validation to the verification work flow is a necessity.

Software model validation is a relatively new area of study, and one that we believe will make model checking-based verification techniques more broadly applicable. For example, one deterrent to formally verifying software systems is the high overhead required for software maintenance. Any updates to a system invalidate the previous verification efforts. Model validation can help with this problem by detecting whether updates to the software will also require updates to the model and, therefore, re-running the verification. In the case that updates to the model are required, a model validation tool could provide hints to the modeler about where in the model the updates are required. We envision a future in which there exist a wide variety of model validation tools, just as today there exist many model checking tools.

Bibliography

[1] Gartner, "Gartner says worldwide public cloud services market to total $131 billion [press release]," February 2013, http://www.gartner.com/newsroom/id/2352816.

[2] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd Symposium on Networked Systems Design & Implementation*. USENIX Association, 2005, pp. 273–286.

[3] C. Chen, P. Maniatis, A. Perrig, A. Vasudevan, and V. Sekar, "Towards verifiable resource accounting for outsourced computation," in *Proceedings of the 9th*. ACM, 2013, pp. 167–178.

[4] P. M. Chen and B. D. Noble, "When virtual is better than real [operating system relocation to virtual machines]," in *Proceedings of the Eigth Workshop on Hot Topics in Operating Systems*. IEEE, 2001, pp. 133–138.

[5] M. Zaharia, S. Katti, C. Grier, V. Paxson, S. Shenker, I. Stoica, and D. Song, "Hypervisors as a foothold for personal computer security: An agenda for the research community," University of California, Berkeley, Tech. Rep., 2012.

[6] P. England, B. Lampson, J. Manferdelli, and B. Willman, "A trusted open platform," *Computer*, vol. 36, no. 7, pp. 55–62, 2003.

[7] P. England and J. Manferdelli, "Virtual machines for enterprise desktop security," *Information Security Technical Report*, vol. 11, no. 4, pp. 193–202, 2006.

[8] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 193–206.

[9] A. Vasudevan, B. Parno, N. Qu, V. D. Gligor, and A. Perrig, "Lockdown: Towards a safe and practical architecture for security applications on commodity platforms," in *Trust and Trustworthy Computing*. Springer, 2012, pp. 34–54.

[10] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2003, pp. 191–206.

[11] K. Kourai and S. Chiba, "HyperSpector: virtual distributed monitoring environments for secure intrusion detection," in *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE)*. ACM, 2005, pp. 197–207.

[12] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through VMM-based out-of-the-box semantic view reconstruction," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2007, pp. 128–138.

[13] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "VMM-based hidden process detection and identification using Lycosid," in *Proceedings of the 2008 ACM International Conference on Virtual Execution Environments (VEE)*. ACM, 2008, pp. 91–100.

[14] L. Litty, H. A. Lagar-Cavilla, and D. Lie, "Hypervisor support for identifying covertly executing binaries." in *USENIX Security Symposium*, 2008, pp. 243–258.

[15] K. Asrigo, L. Litty, and D. Lie, "Using VMM-based sensors to monitor honeypots," in *Proceedings of the 2nd ACM/USENIX International Conference on Virtual Execution Environments (VEE)*. ACM, 2006, pp. 13–23.

[16] J. Chow, T. Garfinkel, and P. M. Chen, "Decoupling dynamic program analysis from execution in virtual environments," in *Proceedings of the USENIX Annual Technical Conference*, 2008, pp. 1–14.

[17] J. R. Crandall, G. Wassermann, D. A. S. Oliveira, Z. Su, S. Felix, W. Frederic, and T. Chong, "Temporal search: Detecting hidden malware timebombs with virtual machines," in *Operating Systems Review (OSR)*. ACM Press, 2006, pp. 25–36.

[18] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2008, pp. 51–62.

[19] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "ReVirt: Enabling intrusion analysis through virtual-machine logging and replay," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*. ACM, 2002, pp. 211–224.

[20] A. Lanzi, M. I. Sharif, and W. Lee, "K-Tracer: A system for extracting kernel malware behavior." in *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2009, pp. 1–16.

[21] D. Quist, L. Liebrock, and J. Neil, "Improving antivirus accuracy with hypervisor assisted analysis," *Journal in Computer Virology*, vol. 7, no. 2, pp. 121–131, 2011.

[22] R. Riley, X. Jiang, and D. Xu, "Multi-aspect profiling of kernel rootkit behavior," in *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys)*. ACM, 2009, pp. 47–60.

[23] A. Vasudevan, N. Qu, and A. Perrig, "XTRec: Secure real-time execution trace recording on commodity platforms," in *Proceedings of the 44th Hawaii International Conference on System Sciences (HICSS)*, Jan. 2011, pp. 1–10.

[24] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging operating systems with time-traveling virtual machines," in *Proceedings of the USENIX Annual Technical Conference*. USENIX, 2005, pp. 1–15.

[25] A. Fattori, R. Paleari, L. Martignoni, and M. Monga, "Dynamic and transparent analysis of commodity production systems," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. IEEE, ACM, 2010, pp. 417–426.

[26] E. Bosman, A. Slowinska, and H. Bos, "Minemu: The worlds fastest taint tracker," in *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID)*. Springer, 2011, pp. 1–20.

[27] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2008, pp. 233–247.

[28] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing," in *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*. Springer, 2008, pp. 1–20.

[29] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2007, pp. 335–350.

[30] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-VM monitoring using hardware virtualization," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2009, pp. 477–487.

[31] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2009, pp. 545–554.

[32] X. Xiong, D. Tian, and P. Liu, "Practical protection of kernel integrity for commodity OS from untrusted extensions." in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2011.

[33] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," in *Proceedings of the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2008, pp. 2–13.

[34] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB reduction and attestation," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2010, pp. 143–158.

[35] R. Ta-Min, L. Litty, and D. Lie, "Splitting interfaces: Making trust between applications and operating systems configurable," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 2006, pp. 279–292.

[36] J. Yang and K. G. Shin, "Using hypervisor to provide data secrecy for user applications on a per-page basis," in *Proceedings of the Fourth ACM International Conference on Virtual Execution Environments (VEE)*. ACM, 2008, pp. 71–80.

[37] Z. Wang, C. Wu, M. Grace, and X. Jiang, "Isolating commodity hosted hypervisors with HyperLock," in *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*. ACM, 2012, pp. 127–140.

[38] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2011, pp. 203–216.

[39] Xen Project, "Xen overview," http://wiki.xen.org/wiki/Xen_Overview, [Online; Accessed: 2013-07-20].

[40] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-Fi tests for Lo-Fi emulators," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2012, pp. 337–348.

[41] L. Martignoni, R. Paleari, G. Roglia, and D. Bruschi, "Testing CPU emulators," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2009, pp. 261–272.

[42] National Vulnerability Database, "CVE-2007-4993," http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-4993, 2007, [Online; Accessed: 2013-07-20].

[43] K. Kortchinsky, "Cloudburst: Hacking 3D (and breaking out of VMware)," in *Blackhat USA*, 2009.

[44] National Vulnerability Database, "CVE-2008-0923," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0923, [Online; Accessed: 2013-07-20].

[45] VMware, "VMware security advisory vmsa-2009-0015," http://www.vmware.com/security/advisories/VMSA-2009-0015.html, 2009.

[46] R. Sinha, C. Sturton, P. Maniatis, S. A. Seshia, and D. Wagner, "Verification with small and short worlds," in *Proceedings of the IEEE Conference on Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2012, pp. 68–77.

[47] C. Sturton, R. Sinha, T. H. Y. Dang, S. Jain, M. McCoyd, W. Y. Tan, P. Maniatis, S. A. Seshia, and D. Wagner, "Symbolic software model validation," in *Proceedings of the 11th ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. ACM, IEEE, 2013.

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

[49] R. P. Goldberg, "Architectural principles for virtual computer systems," Ph.D. dissertation, Harvard University, 1972.

[50] VMware, http://http://www.vmware.com/.

[51] The Xen Hypervisor, http://www.xen.org/.

[52] Kernel-based Virtual Machine, http://www.linux-kvm.org/page/Main_Page.

[53] VirtualBox, https://www.virtualbox.org/.

[54] D. Mihocka and S. Shwartsman, "Virtualization without direct execution or jitting: Designing a portable virtual machine infrastructure," in *The 1st Workshop on Architectural and Microarchitectural Support for Binary Translation*, 2008.

[55] QEMU, http://wiki.qemu.org/Main_Page.

[56] E. M. Clarke and E. A. Emerson, "The design and synthesis of synchronization skeletons using temporal logic," in *Workshop on Logics of Programs*, ser. Springer-Verlag Lecture Notes in Computer Science, 1981, pp. 52–71.

[57] E. A. Emerson and E. M. Clarke, "Using branching time temporal logic to synthesize synchronization skeletons," *Science of Computer programming*, vol. 2, no. 3, pp. 241–266, 1982.

[58] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 2, pp. 244–263, 1986.

[59] J.-P. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR," in *International Symposium on Programming*, ser. Lecture Notes in Computer Science, vol. 137. Springer-Verlag, 1982, pp. 337–351.

[60] A. Pnueli, "The temporal logic of programs," in *Foundations of Computer Science, 1977., 18th Annual Symposium on*. IEEE, 1977, pp. 46–57.

[61] E. A. Emerson, "The beginning of model checking: A personal perspective," in *25 Years of Model Checking*. Springer, 2008, pp. 27–45.

[62] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in computers*, vol. 58, pp. 117–148, 2003.

[63] H. Chen, D. Dean, and D. Wagner, "Model checking one million lines of C code." in *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2004, pp. 171–185.

[64] T. Ball, V. Levin, and S. K. Rajamani, "A decade of software model checking with SLAM," *Communications of the ACM*, vol. 54, no. 7, pp. 68–76, 2011.

[65] P. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson, *A provably secure operating system: The system, its applications, and proofs*. SRI International, 1980.

[66] R. J. Feiertag and P. G. Neumann, "The foundations of a provably secure operating system (PSOS)," in *Proceedings of the National Computer Conference*. AFIPS Press, 1979, pp. 329–334.

[67] B. J. Walker, R. A. Kemmerer, and G. J. Popek, "Specification and verification of the UCLA Unix security kernel," *Communications of the ACM*, vol. 23, no. 2, pp. 118–131, 1980.

[68] W. R. Bevier, "Kit and the short stack," *Journal of Automated Reasoning*, vol. 5, no. 4, pp. 519–530, 1989.

[69] ——, "Kit: A study in operating system verification," *Software Engineering, IEEE Transactions on*, vol. 15, no. 11, pp. 1382–1396, 1989.

[70] J. M. Rushby, "Design and verification of secure systems," in *ACM SIGOPS Operating Systems Review*, vol. 15, no. 5. ACM, 1981, pp. 12–21.

[71] D. Greve, M. Wilding, and W. M. Vanfleet, "A separation kernel formal security policy," in *Proceedings of the Fourth International Workshop on the ACL2 Prover and Its Applications (ACL2)*, 2003.

[72] C. Baumann, T. Bormer, H. Blasum, and S. Tverdyshev, "Proving memory separation in a microkernel by code level verification," in *Proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*. IEEE, 2011, pp. 25–32.

[73] R. J. Richards, "Modeling and security analysis of a commercial real-time operating system kernel," in *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, 2010, pp. 301–322.

[74] W. Martin, P. White, F. Taylor, and A. Goldberg, "Formal construction of the mathematically analyzed separation kernel," in *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering (ASE)*. IEEE, 2000, pp. 133–141.

[75] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean, "Formal specification and verification of data separation in a separation kernel for an embedded system," in *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2006, pp. 346–355.

[76] C. L. Heitmeyer, M. M. Archer, E. I. Leonard, and J. D. McLean, "Applying formal methods to a certifiably secure software system," *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 82–98, 2008.

[77] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Oct. 2009, pp. 207–220.

[78] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "seL4: from general purpose to a proof of information flow enforcement," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (SP)*, 2013.

[79] G. Barthe, G. Betarte, J. D. Campo, and C. Luna, "Formally verifying isolation and availability in an idealized model of virtualization," in *FM 2011: Formal Methods*. Springer, 2011, pp. 231–245.

[80] The Coq Proof Assistant, http://coq.inria.fr/.

[81] J. McDermott and L. Freitas, "A formal security policy for Xenon," in *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering*. ACM, 2008, pp. 43–52.

[82] L. Freitas and J. McDermott, "Formal methods for security in the Xenon hypervisor," *International Journal on Software Tools for Technology Transfer*, vol. 13, no. 5, pp. 463–489, 2011.

[83] J. McDermott, J. Kirby, B. Montrose, T. Johnson, and M. Kang, "Re-engineering Xen internals for higher-assurance security," *Information Security Technical Report*, vol. 13, no. 1, pp. 17–24, 2008.

[84] D. Leinenbach and T. Santen, "Verifying the Microsoft Hyper-V hypervisor with VCC," in *FM 2009: Formal Methods*. Springer, 2009, pp. 806–809.

[85] Hyper-V, http://www.microsoft.com/en-us/server-cloud/hyper-v-server/default.aspx.

[86] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "VCC: A practical system for verifying concurrent C," in *Theorem Proving in Higher Order Logics*. Springer, 2009, pp. 23–42.

[87] E. Alkassar, M. A. Hillebrand, W. Paul, and E. Petrova, "Automated verification of a small hypervisor," in *Verified Software: Theories, Tools, Experiments*. Springer, 2010, pp. 40–54.

[88] E. Alkassar, E. Cohen, M. A. Hillebrand, M. Kovalev, and W. J. Paul, "Verifying shadow page table algorithms," in *Proceedings of the IEEE Conference on Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2010, pp. 267–270.

[89] U. Steinberg and B. Kauer, "NOVA: a microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th ACM European Conference on Computer Systems (EuroSys)*. ACM, 2010, pp. 209–222.

[90] H. Tews, M. Völp, and T. Weber, "Formal memory models for the verification of low-level operating-system code," *Journal of Automated Reasoning*, vol. 42, no. 2–4, pp. 189–227, 2009.

[91] H. Tews, T. Weber, and M. Vlp, "A formal model of memory peculiarities for the verification of low-level operating-system code," in *Proceedings of the 3rd International Workshop on Systems Software Verification (SSV)*. Elsevier, 2008, pp. 79–96.

[92] H. Tews, T. Weber, M. Völp, E. Poll, M. v. Eekelen, and P. v. Rossum, "Nova microhypervisor verification," Radboud University Nijmegen, Tech. Rep. ICIS–R08012, May 2008.

[93] 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 *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2013.

[94] AMD, "Secure virtual machine architecture reference manual," pp. 1–124, 2005, [Online; Accessed: 2013-06-28].

[95] AMD, "AMD-V nested paging, white paper," 2008, accessed June 28, 2013. [Online]. Available: http://sites.amd.com/us/business/it-solutions/virtualization/Pages/amd-v.aspx

[96] M. Gillespie, "Best practices for paravirtualization enhancements from Intel virtualization technology: EPT and VT-d," 2009, accessed June 28, 2013. [Online]. Available: http://software.intel.com/en-us/articles/best-practices-for-paravirtualization-enhancements-from-intel-virtualization-technology-ept-and-vt-d

[97] AMD, "AMD I/O virtualization technology (IOMMU) specification," 2009, accessed June 28, 2013. [Online]. Available: http://support.amd.com/us/Embedded_TechDocs/34434-IOMMU-Rev_1.26_2-11-09.pdf

[98] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert, "Intel virtualization technology for directed I/O," *Intel Technology Journal*, vol. 10, no. 3, pp. 179–192, 2006.

[99] J. Greene, "Intel trusted execution technology: Hardware-based technology for enhancing server platform security, white paper," *Intel*, accessed June 28, 2013. [Online]. Available: http://www.intel.com/content/www/us/en/trusted-execution-technology/trusted-execution-technology-security-paper.html

[100] T. Ormandy, "An empirical study into the security exposure to hosts of hostile virtualized environments," in *CanSecWest*, 2007.

[101] R. Paleari, L. Martignoni, G. Roglia, and D. Bruschi, "N-version disassembly: Differential testing of x86 disassemblers," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2010.

[102] J. Franklin, S. Chaki, A. Datta, J. M. McCune, and A. Vasudevan, "Parametric verification of address space separation," in *Principles of Security and Trust*. Springer, 2012, pp. 51–68.

[103] J. Franklin, S. Chaki, A. Datta, and A. Seshadri, "Scalable parametric verification of secure systems: How to verify reference monitors without worrying about data structure size," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2010, pp. 365–379.

[104] K. S. Namjoshi, "Symmetry and completeness in the analysis of parameterized systems," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Springer, 2007, pp. 299–313.

[105] S. K. Lahiri and R. E. Bryant, "Predicate abstraction with indexed predicates," *ACM Transactions on Computational Logic (TOCL)*, vol. 9, no. 1, 2007.

[106] P. Bjesse, "Word-level sequential memory abstraction for model checking," in *Proceedings of the IEEE Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2008, pp. 1–9.

[107] S. German, "A theory of abstraction for arrays," in *Proceedings of the IEEE Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2011, pp. 299–313.

[108] R. E. Bryant, S. K. Lahiri, and S. A. Seshia, "Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions," in *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*. Springer-Verlag, 2002, pp. 78–92.

[109] A. J. Isles, R. Hojati, and R. K. Brayton, "Computing reachable control states of systems modeled with uninterpreted functions and infinite memory," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, ser. LNCS 1427, A. J. Hu and M. Y. Vardi, Eds. Springer-Verlag, June 1998, pp. 256–267.

[110] R. Kurshan, "Automata-theoretic verification of coordinating processes," in *Proceedings of the 11th International Conference on Analysis and Optimization of Systems – Discrete Event Systems*, G. Cohen and J.-P. Quadrat, Eds. Springer Berlin / Heidelberg, 1994, vol. 199, pp. 16–28.

[111] R. E. Bryant, S. K. Lahiri, and S. A. Seshia, "Convergence testing in term-level bounded model checking," in *Proceedings of the 2003 Advanced Research Working Conference, Correct Hardware Design and Verification Methods (CHARME)*. Springer-Verlag, 2003, pp. 348–362.

[112] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-Into-Libc Without Function Calls (on the x86)," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2007, pp. 552–561.

[113] UCLID, http://uclid.eecs.berkeley.edu/wiki/index.php/Main_Page.

[114] Plingeling SAT Solver, http://fmv.jku.at/lingeling.

[115] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. D. Zuck, "Parameterized verification with automatically computed inductive assertions," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2001, pp. 221–234.

[116] K. L. McMillan and L. D. Zuck, "Invisible invariants and abstract interpretation," in *Proceedings of the 18th International Static Analysis Symposium*, 2011, pp. 249–262.

[117] K. L. McMillan, "Verification of infinite state systems by compositional model checking," in *Correct Hardware Design and Verification Methods (CHARME)*, 1999, pp. 219–234.

[118] M. N. Velev, R. E. Bryant, and A. Jain, "Efficient modeling of memory arrays in symbolic simulation," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 1997, pp. 388–399.

[119] M. K. Ganai, A. Gupta, and P. Ashar, "Efficient modeling of embedded memories in bounded model checking," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, ser. Lecture Notes in Computer Science, vol. 3114, 2004, pp. 440–452.

[120] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, A. Biere, H. van Maaren, and T. Walsh, Eds. IOS Press, 2009, vol. 4, ch. 8.

[121] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[122] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2008, pp. 209–224.

[123] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 2000.

[124] F. Tip, "A survey of program slicing techniques," *Journal of programming languages*, vol. 3, no. 3, pp. 121–189, 1995.

[125] LLVM, http://llvm.org/.

[126] W. H. Harman, "TCAS – a system for preventing midair collisions," *The Lincoln Laboratory Journal*, vol. 2, pp. 437–458, 1989.

[127] J. Kuchar and A. C. Drumm, "The traffic alert and collision avoidance system," *Lincoln Laboratory Journal*, vol. 16, no. 2, p. 277, 2007.

[128] T. Ostrand, "Tcas," Software-artifact Infrastructure Repository, accessed May, 2013. [Online]. Available: http://sir.unl.edu/portal/bios/tcas.php

[129] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and control flow-based test adequacy criteria," in *Proceedings of the 16th International Conference on Software Engineering*, 1994, pp. 191–200.

[130] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 1993, pp. 203–216.

[131] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC architecture," in *Proceedings of the 15th USENIX Security Symposium*, 2006.

[132] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: a sandbox for portable, untrusted x86 native code," *Communications of the ACM*, pp. 91–99, 2010.

[133] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A sandbox for portable, untrusted x86 native code," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2009.

[134] E. Clarke and D. Kroening, "Hardware verification using ANSI-C programs as a reference," in *Proceedings of the 2003 Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE Computer Society Press, 2003, pp. 308–311.

[135] X. Feng and A. J. Hu, "Early cutpoint insertion for high-level software vs. RTL formal combinational equivalence verification," in *Proceedings of the Design Automation Conference (DAC)*. ACM, 2006, pp. 1063–1068.

[136] A. J. Hu, "High-level vs. RTL combinational equivalence: An introduction," in *Proceedings of the 25th IEEE International Conference on Computer Design*. IEEE, 2007, pp. 274–279.

[137] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley, "Solver technology for system-level to RTL equivalence checking," in *Proceedings of the 2009 Design, Automation and Test in Europe (DATE)*, 2009, pp. 196–201.

[138] A. Koelbl, J. R. Burch, and C. Pixley, "Memory modeling in ESL–RTL equivalence checking," in *Proceedings of the Design Automation Conference (DAC)*. IEEE, 2007, pp. 205–209.

[139] A. Koelbl and C. Pixley, "Constructing efficient formal models from high-level descriptions using symbolic simulation," *International Journal of Parallel Programming*, vol. 33, no. 6, pp. 645–666, 2005.

[140] M. Hind and A. Pioli, "Which pointer analysis should I use?" in *ACM SIGSOFT Software Engineering Notes*, vol. 25, no. 5. ACM, 2000, pp. 113–123.

[141] B. Hardekopf and C. Lin, "Flow-sensitive pointer analysis for millions of lines of code," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2011, pp. 289–298.

[142] J. Caballero, S. McCamant, A. Barth, and D. Song, "Extracting models of security-sensitive operations using string-enhanced white-box exploration on binaries," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-36, Mar 2009.

[143] D. A. Ramos and D. R. Engler, "Practical, low-effort equivalence verification of real code," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*. Springer, 2011, pp. 669–685.

[144] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke, "Using model checking with symbolic execution to verify parallel numerical programs," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2006, pp. 157–168.

[145] ——, "Combining symbolic execution with model checking to verify parallel numerical programs," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 17, no. 2, p. 10, 2008.

[146] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, "Differential symbolic execution," in *Proceedings of the 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM, 2008, pp. 226–237.