Techniques for Model Inference and Bug Finding

by

Chia Yuan Cho

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 Dawn Xiaodong Song, Chair
Professor Sanjit A. Seshia
Professor John Chuang

Fall 2013

Techniques for Model Inference and Bug Finding

Abstract

Techniques for Model Inference and Bug Finding

by

Chia Yuan Cho

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Dawn Xiaodong Song, Chair

Security bugs in network-based applications allow an attacker to compromise a system from the network. Existing techniques to find bugs in network-based applications are limited because the rules by which a program interacts with its network environment (i.e., the protocol) are often unknown. Further, the complexity of stateful interactions in such applications makes bug-finding difficult.

In this thesis, we explore two directions towards solving the problem of finding bugs in network-based applications. First, to overcome the problem of not knowing how an application interacts with its network environment, we propose new techniques to automatically infer the protocol model implemented by an application, and use the inferred model to guide the search for bugs. We apply our technique to infer the command-and-control protocol model of a major spam botnet, and analyze it to discover its weakest link and protocol logic flaws. The analysis leads to new ways to disable botnets and new ways to fight spam.

Second, we propose new techniques to analyze programs to find bugs. Our techniques are based on a family of techniques called symbolic analysis, that are equivalent to testing with entire classes of test case inputs. We show that a synergistic combination of model inference and symbolic analysis is more effective than symbolic analysis alone. Our technique finds several new vulnerabilities in popular applications that have remained undetected for years.

Symbolic analysis has limited scalability on programs of significant sizes. We propose a compositional approach to symbolic analysis, that scales to programs in the order of hundreds of thousands of lines of code (100 KLOC). Our technique uses a novel combination of satisfying assignments and proofs of unsatisfiability in a solver to generalize preconditions that must lead to bugs. Using the preconditions with the data-flow of the program allows it to prune irrelevant code from analysis. Our technique outperforms existing tools, and finds multiple new vulnerabilities in critical Internet infrastructure software.

In memory of Mom

Contents

List of Figures

List of Tables

Acknowledgments

This thesis has benefited from collaborations and conversations with many people. First and foremost is my advisor Dawn Song. She accepted an irresponsible student like me into her group and gave me tremendous autonomy to go after my research interests. She has great energy and contagious passion for scientific research, putting together some of the best minds in diverse fields into her group. The synergy of ideas has been stimulating and rewarding.

My thesis committee has helped me more than I can ever thank them. I am grateful to Sanjit Seshia for helping me build a foundation in formal verification. One of the concepts that he talked about kept me intrigued and led me to develop the compositional bounded model checking technique in this thesis. John Chuang gave me a refreshing economics and business perspective into technology. He inspired me to learn more in these areas and grow more broadly. I am also indebted to Vern Paxson for being my occasional mentor and for his insightful guidance, and David Wagner for thoughtful comments that helped improve my work.

I have also learned a lot from my colleagues and collaborators. I was privileged to work with Vijay D'Silva and Domagoj Babic in particular. Vijay is a combination of deep knowledge with light-hearted demeanor; Domagoj's work ethic is legendary. His knowledge on grammar inference led to our collaborations on adapting it to botnet and program analysis. I have also benefited from working or having conversations with Stephen McCamant, Juan Caballero, Pongsin Poosankam, Prateek Saxena, Devdatta Akhawe, Kevin Zhijie Chen, Noah Johnson, Chris Grier, Mathias Payer, Mario Frank, Richard Shin and Edward XueJun Wu. Thank you all for enriching my academic life.

My family has provided me with inspiration and support every step of my journey. My wife is my indispensable life companion. She improves my strengths and complements my weaknesses (I have many), and she knows I am helpless without her. My Mom showed me the virtues of hard work and dedication, and my Dad impressed me with his ingenuity (when I was a kid). I am the result of their work. In the same way, I hope to be a great Dad (learning in process) to my sons Aidan and Dylan, who have brought unimaginable joy to our lives. I am also grateful to my sister and brother for taking care of the family, making up for my long absence. Finally, thanks to my friends in Berkeley and Singapore. You have all brightened up my life.

Chapter 1

Introduction

We rely on software applications for our computing and communication needs over the Internet. However, these network-based applications also provide the bridge between a system and an attacker from the network. A network-based application takes untrusted input from its network environment. Any bug in the application that can be triggered by an untrusted input is a potential security vulnerability that may be exploited remotely by an attacker. Application developers have always aimed to find and fix such errors before shipping the software. Their success has been limited so far, because network-based applications are complex and prone to bugs, but the techniques to find bugs are limited. In this thesis, we seek to answer this question: How can we overcome the problem of finding bugs automatically in network-based applications?

Two challenges characterize the problem. First, network-based applications are complex stateful programs that maintain an ongoing interaction with their network environment, rather than processing some input to produce some final output upon termination[1]. The complexity of stateful interactions makes the task of finding bugs more difficult — detecting protocol logic errors requires reasoning on software behavior at the level of protocol logic, while low-level implementation errors are often deep and not easily reachable. Second, the actual rules by which a program interacts with its network environment (i.e., the protocol) are often unknown. For example, the program may be using a closed or proprietary protocol (e.g., in malware). Even on open protocols, implementations differ in undocumented ways from open standards. Because the actual protocol implemented by a complex application is often not fully known even to its own authors, in practice developers find vulnerabilities in their own code in a black-box manner by testing their application with randomly generated inputs, i.e., black-box fuzz testing. For example, the Google Chrome development team has built a dedicated cluster called ClusterFuzz to fuzz-test the Chrome browser with 50M test cases a day[2].

More broadly, existing bug-finding techniques generally fall into three categories: testing (dynamic analysis), static analysis, and model checking. Since finding bugs in software is undecidable in general — a result of the Halting Problem, all automated techniques must either miss bugs (false negatives), or report bugs that do not actually exist (false positives). Testing inherently misses bugs,

[1] In other words, network-based applications are reactive systems.

[2] `http://blog.chromium.org/2012/04/fuzzing-for-security.html`

since the number of execution paths in any non-trivial program is astronomically large or infinite. Sound static analyses cope with this problem by computing sound abstractions of program behavior, but the limits in precision cause false positives. Symbolic model checking techniques also compute sound abstractions, but emphasize improving the precision over successive abstraction refinements to avoid false positives. However, because the number of refinements for software can be unbounded, model checkers may run into memory exhaustion, and may not terminate. Despite the enormous potential of static analysis and model checking, testing is currently by far the most commonly used technique to find bugs in practice. The New Electronics 2013 survey[3] found that 88% of developers use manual testing and debugging, but only 34% use static analyzers, and none use model checkers. Clearly, we need to equip developers with better techniques and more powerful tools to find bugs. More importantly, we believe that developers' preference for testing is rooted in extremely low tolerance for false positives — they would rather miss bugs than waste time investigating false positives. Therefore, if we can develop bug-finding techniques that are much more powerful than testing, but preserve its no-false-positive property, then we can be reasonably confident that such techniques will be useful for developers. This is the overarching principle for all techniques that we propose.

In this thesis, we explore two directions towards solving the problem of finding bugs in network-based applications. First, to overcome the problem of not knowing how an application interacts with its network environment, we propose new techniques to infer the protocol model implemented by an application, and use the inferred model to guide the search for bugs. To deal with the practical setting of sometimes not having direct access to the application (e.g., a botnet's Command-and-Control servers), we propose a black-box technique to infer an application's protocol model remotely over the network. We show that active automata learning algorithms can be successfully applied to remotely infer the protocol model implemented by an application. We apply our technique to infer the Command-and-Control (C&C) protocol model of a major spam botnet, and analyze it to discover its weakest link, protocol logic flaws and unobservable communication between C&C servers. The weakest link analysis leads to potentially more efficient ways to take down and disable a botnet, and the discovered protocol logic flaws enable new ways to fight spam.

Second, we propose new techniques to analyze programs to find bugs. Like testing, the techniques that we propose are free of false positives, i.e., any found bug is a true bug. Unlike testing, our techniques are based on the symbolic analysis of programs, achieving the same effect as testing with entire classes of test cases. We show that model inference and symbolic analysis can be combined in a synergistic manner. Model inference computes a model of program behavior to guide the search for bugs, while new program behavior discovered during symbolic analysis refines the inferred model. The two sub-components alternate till convergence to fixed point. Using such a synergistic combination of model inference and symbolic analysis, our technique discovered seven unique vulnerabilities in widely-used desktop applications, four of which are new and filed in the National Vulnerability Database.

[3] http://www.newelectronics.co.uk/electronics-technology/ne-embedded-survey-results/47463/

Symbolic analysis can be viewed as a bounded model checking (BMC) technique, and shares the same limitation of running out of memory or time on real-world programs of significant sizes. We propose a new compositional bounded model checking technique, that scales to large programs in the order of hundreds of thousands of lines of code (100 KLOC). Our technique is compositional — it decomposes a program into small code fragments, checks a single fragment at a time to compute a precondition that leads to a bug, and propagates the precondition across fragments towards the program entry point. Our technique uses a novel combination of satisfying assignments and proofs of unsatisfiability in a solver to: (1) generalize preconditions that lead to the bug, and (2) identify only the code fragments that are relevant to the bug, bypassing irrelevant code. Our technique outperforms existing tools on a set of benchmarks. Using the technique to analyze critical Internet infrastructure software used by Internet Service Providers worldwide, we discovered multiple new vulnerabilities.

1.1 Contributions

This thesis makes the following contributions.

Chapter 3: Inferring Protocol Models over the Network. We propose a technique to infer the protocol model of an application remotely over the network. We apply our technique to infer the command-and-control (C&C) protocol model of a spam botnet, and analyze it to discover its protocol logic flaws, weakest link and unobservable communication channels between C&C servers. The analysis leads to new ways to disable botnets and new ways to fight spam [20].

Chapter 4: Synergies between Model Inference and Symbolic Analysis. We propose MACE, a technique that synergizes model inference and symbolic analysis to discover both the applications's protocol model and security bugs. Experimental results show that MACE explores more deeply into a program, discovering seven vulnerabilities in widely-used desktop applications [19].

Chapter 5: Compositional Bounded Model Checking for Real-world Programs. We propose BLITZ, a compositional and property-sensitive algorithm that enables bounded model checking to automatically find bugs in large programs. BLITZ scales to real-world programs with at least 100K lines of code, outperforming existing tools and discovering multiple new vulnerabilities in critical Internet infrastructure software [22].

Chapter 2

Background and Overview

In this thesis, we propose techniques to automatically discover vulnerabilities in network-based applications. We will first propose techniques to automatically infer the protocol model from the implementation of a network-based applications, and use that to guide the search for bugs. We will then propose new techniques to analyze programs to find bugs, that improve the effectiveness and scalability of existing techniques. In Section 2.1, we provide examples and definitions of vulnerabilities in network-based applications. In Sections 2.2 and 2.3, we introduce background and basic terminology on the model inference and bug-finding techniques that we propose in this thesis.

2.1 Vulnerabilities in Network-based Applications

Network-based applications can be written in almost any programming language, but C/C++ remains most popular today[1]. Low level languages like C/C++ are inherently memory unsafe and susceptible to memory corruption errors, which are exploited by attackers to alter or gain control of a program's control flow. For example, an attacker with control over an array index can cause the program to read or write outside the array bounds. Consider the following code fragment where a network input is used as an array index into a jump table without bounds checking. The attacker can divert control flow to any memory location under his control.

```
input = get_network_input();
jump_table[input]();
```

The above vulnerability is an example of an *indexing bug*. An attacker may also exploit other memory errors such as *buffer overflow*, *null pointer dereference*, *use-after-free* and *format string* to hijack control flow[2]. These bugs are collectively known as memory safety violations, the class of critical low level bugs that our techniques seek to find in this thesis. It is relatively easy to

[1] TIOBE Programming Community Index, `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

[2] See [87] for a more complete discussion on memory errors and attacks.

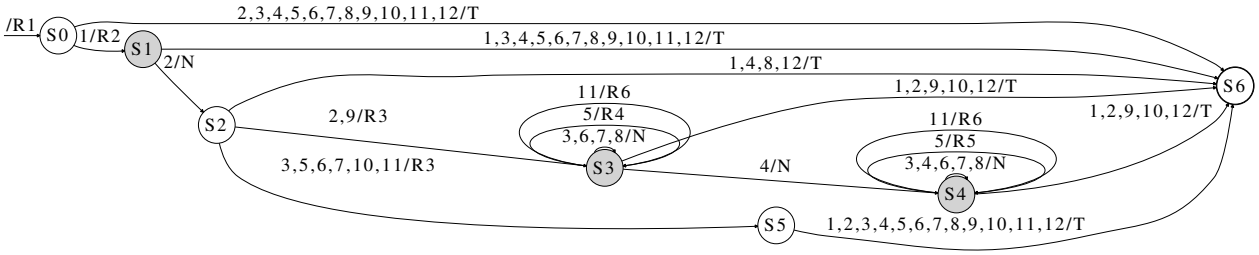


Figure 2.1: RFB Protocol Model in Vino 2.26.1, the default remote desktop in GNOME. States with vulnerabilities are shown in gray. The edge labels show the list of input messages and the corresponding output message separated by the '/' symbol. The explanations of Input/Output Messages of the state machine is shown in Figure 4.4.

specify memory safety properties and insert them into code as static or run-time checks. For example, inserting `assert(input >= 0 && input < SIZE(jump_table));` at line 2 of the above example, where `SIZE(A)` denotes the number of elements in array `A`, we can check if the assertion violation is reachable statically or dynamically. Determining whether a property violation is reachable is an undecidable problem. In the context of network-based applications, which are stateful, bugs can be buried within deep protocol states that are difficult to reach from the initial state.

Network-based applications are also prone to bugs at the level of the application's protocol logic. For example, an application typically requires authentication to grant access to resources. But if there exists an alternative path in the protocol implementation to obtain access to the same resources without authentication, then the application has a protocol logic bug. Like memory safety bugs, some protocol logic bugs allow an attacker to hijack control flow. Unlike memory safety properties, protocol logic properties are application-specific and not as easy to specify. To detect these bugs, we need to reason at the level of the application's protocol logic. However, the protocol implemented by an application is often unknown, even if the application purports to implement an open protocol with known specifications. Industrial black-box fuzzers deal with this problem by requiring the tester to manually specify the protocol logic, which is tedious and may not be accurate. Our approach is to automatically infer the protocol model implemented by an application and use it to guide the search for bugs. We also propose techniques that are more powerful than testing.

We illustrate the challenges of finding bugs in networked applications with an example. Fig. 2.1 shows the model of the RFB (Remote Frame Buffer) protocol implemented in GNOME Vino 2.26.1, the default remote desktop application in GNOME that we will revisit in Chapter 4. The RFB protocol is stateful. It begins at an initial state (S0), and terminates at a terminal state (S6). Multiple vulnerabilities exist in Vino 2.26.1, shown as gray states in Fig. 2.1.

Memory safety bug. States S3 and S4 contain out-of-bounds memory access (i.e., indexing error) vulnerabilities (CVE-2011-0904 and CVE-2011-0905 respectively) — the application uses fields from received network messages as the indices to an array without checking. A remote attacker can craft a malicious RFB message to exploit the application in either of these states.

Protocol logic bug. State S1 contains a Denial-of-Service vulnerability (CVE-2011-0906) due to a protocol logic error — when the application receives an unexpected message in this state, it enters an infinite loop, using up all processor cycles and causing the application to be unresponsive. Obviously, the infinite loop is not consistent with the RFB protocol standard.

The above three vulnerabilities have existed since 2004 and remain undiscovered till seven years later (using our techniques). If effective techniques exist to find bugs automatically in network-based applications, then such vulnerabilities would have been found easily and fixed before they affect the users of GNOME worldwide.

In this thesis, we propose techniques to discover such vulnerabilities automatically in network-based applications. We explore two directions towards solving the problem of finding bugs in network-based applications. First, we propose techniques to automatically infer the protocol model from the implementation of a network-based applications, to guide the search for bugs. Second, we propose new techniques to analyze programs to find bugs, improving the effectiveness and scalability of existing techniques.

2.2 Techniques to Infer Protocol Models

Network protocols are usually modeled as finite-state machines, so we cast the problem of inferring protocol models as an automata learning problem. Automata learning works by observing a set of input/output traces to infer an automata that is both consistent with all observed positive traces, and also rejects all negative traces that cannot be produced by the target automata [45]. All automata learning approaches are either passive approaches that observe traces offline, or active approaches that observe traces online. Because active approaches have control over the traces that they observe by making queries, polynomial algorithms exist to learn complete (and minimal) automata using an active approach [4]. In contrast, an passive approach is inherently incomplete. In this thesis, we choose to develop model inference techniques based on the active approach, so that the protocol models that we infer are complete for a given message set. Prior to our work, the active approach to protocol model inference has received little attention, and most inferred protocol models were incomplete, with some unspecified (missing) transition edges [48, 27].

L^* is the first automata learning algorithm using an active approach [4]. It has found numerous applications in formal verification, such as learning environment assumptions for verification through assume-guarantee reasoning [26, 43, 42], and remains the most widely-used active approach. L^* works by making (membership and equivalent) queries over an input alphabet, and records its observations into an Observation Table [4, 82]. L^* assumes the input and output alphabets are known, which may not be true when applied to protocol model inference. In our setting, the input (output) alphabets correspond to distinct input (output) message types in the protocol, which we often have limited knowledge of. For example, a botnet protocol might be proprietary, and we do not have access to the server-side of botnets. We overcome this limitation in our techniques as follows: In a black-box setting where we do not have access to the software, we first assume all messages are distinct in the alphabet, and then reduce redundancy in our queries by predicting output responses from past observations (Chapter 3 [20]); in a white-box setting, we

propose a technique to automatically discover new inputs in the alphabet (Chapter 4 [19]).

2.3 Techniques to Find Bugs

The problem of finding bugs in software can be formulated as a *reachability* problem. A program is a control flow graph over a set of states and state transitions. An error (bug) is a violation of a property in the program. Given an initial state and an error state, the *error reachability* problem is to determine if there exists a path from the initial state to the error state. We say that an error reachability technique is sound if whenever the technique reports an error, the error is reachable (no false positives). An error reachability technique is complete if whenever the technique reports that the error is not reachable the error is indeed not reachable (no false negatives). Our definitions of soundness and completeness are given with respect to reachability and differ from the notions used in the correctness literature. The error reachability problem is undecidable for software, i.e., there does not exist an error reachability technique that is both sound and complete. Thus any bug-finding technique is either unsound or incomplete (or both). Existing techniques for automated analysis of software generally fall into three categories: testing (dynamic analysis), static analysis, and model checking. We discuss each approach, and extract insights to motivate the analysis techniques that we develop in our proposed approach.

Testing. Testing is the simplest and most commonly used method to find bugs. Fuzz testing in particular is currently the most common approach to find security vulnerabilities — most reported vulnerabilities in the National Vulnerability Database[3] have been discovered through *fuzz testing*. Fuzz testing is a technique that treats a program as a black box, sending randomly generated test cases into it, and detects a fault when the program behaves in an unexpected way (e.g., crash with segmentation fault). It uses test cases that are either randomly generated, or partially mutated from valid test cases [49]. For network-based applications, fuzz testing with awareness of the protocol model is more effective than random testing. However, existing protocol models are either manually constructed[4], or inferred incompletely from network traces [27]. Testing inherently reports no false positives, but reported errors are far from complete (has false negatives). This is because each test case only exercises a single execution path, but programs are infinite-state systems.

Dynamic analysis techniques improve the completeness of conventional testing through code instrumentation. The instrumented code can be added during compilation (compile-time instrumentation, e.g., [71, 58, 81]), or during runtime through a virtual machine (dynamic binary instrumentation, e.g., [60, 72, 12]). Dynamic analysis improve completeness through higher code coverage, or better error detection mechanisms. For example, *automatic test case generation* aims to explore the program state-space systematically to improve code coverage [16, 80, 34]. *Taint-tracking* [86] introduces taint to track the propagation of data from source (e.g., a network input)

[3] `http://nvd.nist.gov/`

[4] `http://peachfuzzer.com/`

to sink (e.g., a system call), to detect critical data (e.g., array index) that originates from the attacker. All dynamic analysis techniques are ultimately based on testing, and share the same error reachability properties.

Static analysis. Static analysis was first proposed for compiler optimization, and later applied to check program correctness. Static analysis computes information on the behavior of a program without actually executing it. Since determining the behavior of a program is undecidable in general, static analysis techniques compute abstractions that over-approximate program behavior. Unlike testing, static analysis can provide strong guarantees on program correctness — if all error states are not reachable despite having over-approximated program behavior, then these errors can *never* occur (i.e., no false negatives). However, any detected error may be a false positive since it could have been introduced by over-approximated behavior that does not exist in the program. Thus, its error reachability report is unsound. For example, consider the following code where variable `i` may take the concrete values {`0, 2, 4`} at line 2. A value-set analysis would over-approximate the concrete values at this location using the superset [0, 4], and wrongly report the assertion violation at line 2 is reachable.

```
1: for(int i=0; i<5; i+=2)
2:     assert(i!=1);
```

Static analysis sacrifice precision for scalability by ignoring some aspects of the program's structure when it computes value sets. A context sensitive analysis takes into account the different call sites of a procedure; a flow sensitive analysis considers the sequential order of executed statements; a path sensitive analysis differentiates between different execution paths. A {context, flow, path}-insensitive analysis is more scalable than a {context, flow, path}-sensitive analysis, but may also report more false positives.

Despite the ability of static analysis to provide strong guarantees on a program's correctness, only 34% of developers use static analysis. This is because false positives impose the burden of manually investigating *all* reported bugs. In this thesis, we use static analysis to compute initial over-approximations of program behavior, and then refine the over-approximations iteratively, similar to the abstraction-refinement frameworks in model checking.

Model checking. Model checking was first proposed for checking the correctness of finite-state systems (e.g., hardware designs) [25]. At a high level, model checking checks a program's correctness by exploring its reachable states exhaustively. If all error states are not reachable, a program is guaranteed to be correct (i.e., no false negatives). Otherwise, the model checker generates a concrete error *witness* from the initial state to the error state. A programmer can use the witness to reproduce the error, just like test cases in dynamic analysis.

Like static analysis, symbolic model checking techniques use abstraction to represent the program state-space, and the two approaches have been shown to be equivalent. Unlike static analysis, model checking also emphasizes precision, required to generate a witness when a bug is found. For example, in the CEGAR (Counter-Example Guided Abstraction Refinement) framework for model

checking [24], an initially coarse abstraction is refined iteratively as needed until it is sufficiently precise to either prove that a property holds for all possible executions, or show that the error exists with a witness. Thus, unlike static analysis, model checking also aims to report no false positives.

Recall that the error reachability problem is undecidable for software, i.e., there does not exist an error reachability technique that is both sound (no false positives) and complete (no false negatives). Since programs are *infinite*-state systems, the required number of refinement iterations may grow indefinitely. Further, each refinement becomes increasingly more expensive to compute. Therefore, software model checkers have been most successfully applied to software in the form of device drivers and embedded systems (e.g., [6, 44, 23]). On larger programs, model checkers tend to run into memory exhaustion, or take too much time to terminate. These are the sub-challenges that we seek to overcome in this thesis.

We summarize our key insights on the above approaches. Static analysis is scalable and can provide strong guarantees on program correctness, but over-approximating program behavior introduces false positives that are undesirable to developers; model checking eliminates false positives by refining abstractions iteratively, but ultimately runs into timeout or memory exhaustion; testing misses bugs but is most commonly used in practice, because it is simple and has no false positives. Clearly, programmers are more willing to miss bugs rather than use a tool that reports false positives.

Guided by this insight, the bug-finding techniques that we develop in this thesis can be seen as more powerful forms of testing — like testing, they have no false positives; unlike testing, they achieve the same effect as testing a program with entire classes of test case inputs. Further, we propose techniques that are scalable and effective without running into memory exhaustion. We discuss the basic techniques that we rely and improve on — *symbolic execution*, and *software bounded model checking*, and provide the intuition to our approaches at a high level. We refer to both techniques as variants of *symbolic analysis* in this thesis.

Symbolic Execution. Symbolic execution was proposed in 1976 [52], but did not take off till decades later. Symbolic execution works by building a path constraint formula for each execution path that all program inputs down that path must satisfy. Using the formula, a property is checked by solving the formula conjoined with the negation of the checked property. A true property violation is found if a solution exists for the formula (i.e., it is satisfiable). The solution is also a concrete error witness that triggers that violation. For example, consider the example code and its control flow graph in Figure 2.2. The assertion at line 4 ensures the buffer `buf` is never written out of bounds. There are multiple paths to this assertion and shortest is the path $3 \rightarrow 4$. The assertion is checked on this path using the formula $(i < 4) \wedge \neg(i >= 0 \wedge i < 5)$. Representing `i` as a signed 32-bit integer and solving the formula, one satisfiable value is `i == -1`, which is an input to the procedure that would trigger the assertion violation.

In the seminal 1976 paper, symbolic execution was envisioned as a powerful "enhanced testing technique" because it achieves the same effect as executing a program on the entire class of inputs down each path, instead of a single (concrete) test case. For instance, if a formula is unsatisfiable,

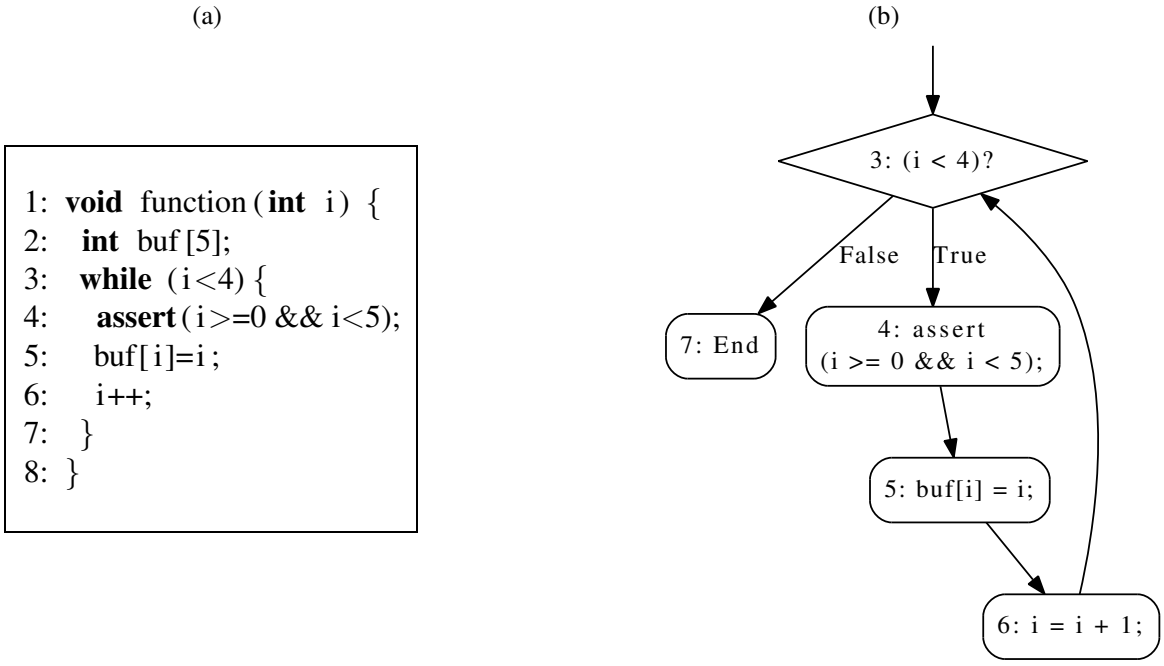


Figure 2.2: An Example Procedure (left) and its Control Flow Graph (right)

then we can immediately conclude that all errors are unreachable along that path (i.e., that path is correct). Using testing, the same conclusion can only be reached by testing exhaustively with all test cases that drive execution along that path.

Symbolic execution requires an efficient constraint solver to solve the formulas. This was thought to be not practical for decades because the (boolean satisfiability) problem is NP-Complete. Contrary to theoretical limitations, the efficiency of boolean satisfiability (SAT) solvers has improved by leaps and bounds in the last decade [62]. It is now widely accepted that although SAT may incur exponential complexity in the worst case, some heuristics do work well for typical formulae encountered in practice. As a result, solvers have been applied increasingly to program analysis. For program state-state exploration, it is now routine to use symbolic execution in combination with concrete execution (testing), where typically an execution is concretized whenever a formula is too hard to solve [77, 51, 80, 16, 34]. However, important challenges remain. Although symbolic execution provides a directed and systematic approach to testing, the number of execution paths (and formulae to solve) remains inhibitively large (and may be infinite). Therefore, coverage is usually still far from complete. To see this, consider Figure 2.3, which depicts the symbolic execution tree built while analyzing the example in Figure 2.2. Since symbolic execution forks whenever a branch condition is encountered and both branch conditions are feasible, it needs to fork (and solve) into up to 2^n paths when n branches are encountered during execution. Since the number of branch conditions in software is usually large (potentially infinite), only a very small number of paths can be ever explored. Further, it gets increasingly difficult to solve path formulas with the increasing depth of exploration. In Chapter 4, we overcome this problem

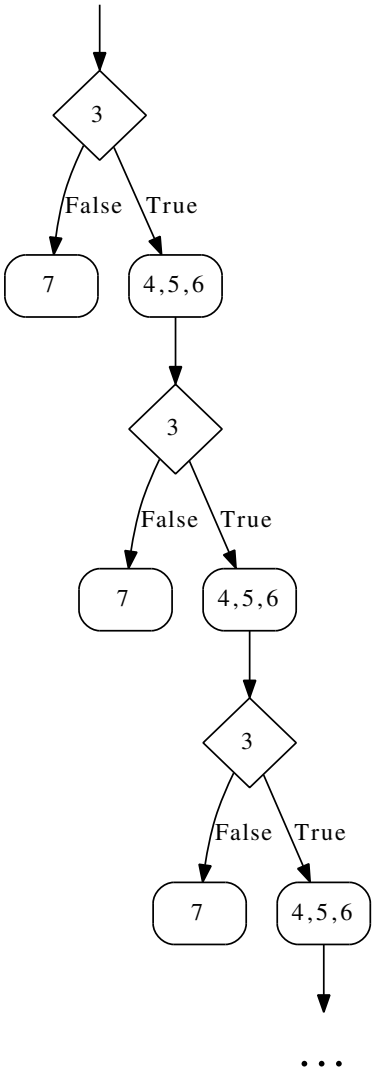


Figure 2.3: A Symbolic Execution Tree for the Example in Figure 2.2

by proposing techniques for *localized* symbolic analysis, but guide that search using a program's inferred protocol model, which provides a *global* view of the program.

Software Bounded Model Checking (BMC). Bounded Model Checking is closely linked to symbolic execution. The rise of SAT solvers was equally embraced in the hardware formal verification community, which adapted symbolic execution to check hardware circuits. This later became a technique known as Bounded Model Checking (BMC) [23]. Applied to software, BMC works by performing a variant of symbolic execution over the program control flow, unwinding all loops and recursion up to some predetermined bound k. Like symbolic execution, BMC is sound and precise, and may not explore the program state-space completely. However, unlike symbolic execution, BMC explores all paths (*guarantees* complete path coverage) up to that bound. In other words, it is a sound and complete technique within the specified bound, and can be used to show bounded correctness.

One key difference between software BMC and symbolic execution is how branch conditions are handled — symbolic execution forks on branches, producing a symbolic execution tree; in BMC, all branches are merged into a *single formula* that describes all paths up to the bound. To retain soundness, all execution paths beyond the bound are cut off using an *assume* condition. For example, the unwinding of the example code in Figure 2.2 with $k = 2$ is as follows.

```
if (i<4) {
  ...
  if (i<4) {
    ...
    if (i<4)
      assume(False);
  }
}
```

The BMC approach of merging all bounded paths into a single formula blows up the formula size and complexity, making it more likely for a solver to run into timeout or memory exhaustion. In Chapter 5, we propose the first compositional technique for BMC, allowing BMC to scale to large programs in the order of 100 KLOC [22].

Chapter 3

Inferring Protocol Models over the Network

3.1 Introduction

Protocol inference is a process of learning the inner workings of a protocol through passive observation of the exchanged messages or through active probing of the agents involved in the message exchange. Even for small simple protocols, manual protocol inference (a.k.a. reverse-engineering) is tedious, error-prone, and time-consuming. Automatic protocol inference sounds attractive, but poses a number of technical challenges, for some of which we propose novel solutions in this thesis.

The applications of protocol inference are numerous. The main application we are interested in is the inference of botnet protocols. Botnets are the primary means through which denial of service attacks, theft of personal data, and spamming are committed, causing billions of dollars of damage annually [1]. Defeating such botnets requires the understanding of their inner workings, i.e., their communication protocols. The second application we are interested in is inferring models of implementations of frequently used protocols. While public standards are often available for such protocols, implementations rarely strictly follow the standard, either due to bugs in the implementation, pitfalls in understanding the standard, or ambiguities in the standard itself. In this setting, automatic protocol inference technology can be used for fingerprinting and checking adherence to the standard. Other possible applications of protocol inference include: automatic abstraction of agents participating in message exchange for assume-guarantee-style verification of protocols, fuzz testing of protocol implementations (e.g., [27]), and reverse-engineering of proprietary and classified protocols.

Our main motivation is to provide the security community with new techniques and tools to fight botnets. The technology we have developed is a powerful weapon against botnets, enabling automatic inference of protocol models that can be used by both human analysts and automatic tools. After presenting our main technical contribution, we propose a number of analyses of inferred models: identification of protocol components most susceptible to disruptive attacks (e.g., for the purpose of finding the most efficient way of bringing down a botnet), identification of protocol design flaws, detection of back-channel communication to which the analyst has no direct

access, and detection of differences among protocol implementations.

A Brief Overview of Protocol Inference

The inner workings of a protocol can be modelled in a number of ways, but state-machine models are by far the most standard. Thus, the problem of protocol inference is isomorphic to the problem of learning a state-machine describing the protocol. Modelling protocols with finite and infinite state-machines is a complex topic, but only two dimensions of the problem are relevant to our work. The first dimension is the *finiteness* of the state-machine. In this thesis, we focus on inferring finite state-machine models, both in terms of the number of states and the size of the input (resp. output) alphabet. The second dimension is the *completeness of the state-machine*, a concept originating in the theory of automata. A complete state-machine has transitions (resp. outputs) defined for every input alphabet symbol and from every state. An incomplete state-machine might be missing some transitions, i.e., for some state-input pairs, the next state (resp. output) is undefined.

It is important to distinguish the automata-theoretic concept of state-machine completeness from the model completeness. Consider a simple protocol Π with input messages $\{a,b,c\}$ and output messages $\{x,y,w\}$, specified by the state-machine in Figure 3.1a. Suppose we have to reverse-engineer the protocol from an implementation of Π while treating it as a black-box. The first step in the inference is to discover the valid input messages. To our knowledge, there exists no automatic approach guaranteeing to discover all the input messages. Suppose a subset M of input messages is discovered during the inference, e.g., $M = \{a,b\}$. An automatic protocol inference approach learns a complete state-machine (Figure 3.1b) with respect to M if from every state of the learned state-machine and for every message from M, the next state and output are known. Otherwise, the inferred state-machine is incomplete (Figure 3.1c). The completeness of state-machine models is critical for aforementioned applications, because the more transitions are missing from the model, the higher the uncertainty of the analysis. In this work, we focus on inferring complete state-machines through interactive *on-line* inference, in contrast to the previous work [27, 48], which focused on passive *off-line* inference of incomplete models. Given a complete alphabet (i.e., set of input messages), our approach infers complete finite-state protocol models with desired accuracy and confidence (the higher the accuracy and confidence, the higher the computational cost).

Passive off-line inference techniques can only learn from a sequence of observed messages. Off-line inference of a minimal (or within a polynomial size of minimal) state-machine from observed network communication is a computationally difficult problem (NP-complete [37], [88, p. 98–99]). Heuristic best-effort inference algorithms are polynomial in the number and size of observed message sequences, as in work of Hsu et al. [48], but there are no guarantees that the inferred model will be minimal, or even at most polynomially larger than the minimal. While off-line techniques can infer models from a relatively small number of observed message sequences, such models are inherently incomplete, rendering any further analysis imprecise.

In contrast, on-line inference techniques are allowed to query the agents involved in the message exchange proactively. On-line techniques, like Angluin's $L*$ algorithm [4], have a polynomial worst-case complexity and produce complete models. Despite polynomial complexity, a number of

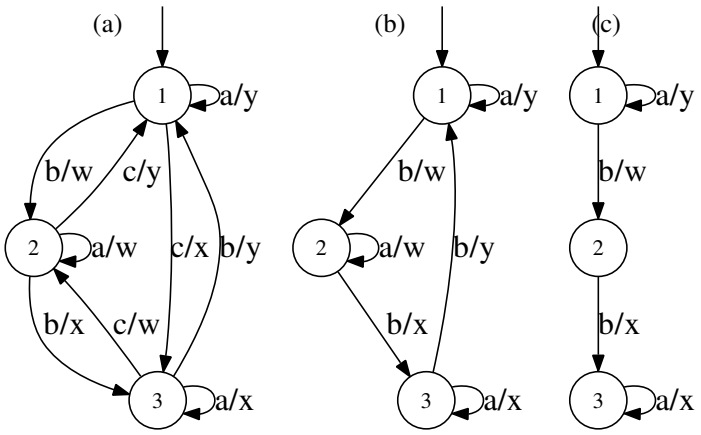


Figure 3.1: Illustration of State Machine Completeness. A complete model with respect to the entire protocol alphabet (on the left), complete state-machine with respect to a given subset of messages (in the middle), and an incomplete state-machine (on the right).

challenges, especially in the context of botnet protocol inference, remain: (1) Automatic inference of complete protocol state-machines in the real-world network setting requires solving a number of subtle technical challenges, ranging from theoretical ones, like choosing the right formal model, to technical ones, like reverse-engineering message formats. (2) Even a simple protocol with twenty states might require tens of thousands of message sequences to be generated. (3) In order to avoid synchronized attacks by a large botnet on the university infrastructure, the experiments had to be anonymized by tunneling all the traffic through Tor [32]. (4) Exacerbated by the usage of Tor and likely overloading of botnet servers, the network delay averaged 6.8 seconds per message in our experiments, dramatically reducing the number of probing sequences feasible in reasonable amount of time.

Main Contributions

On the inference side, the most important contribution of this work is a novel approach for complete state-machine inference in the realistic high-latency network setting. Three innovations rendered such inference possible: First, our formulation of the protocol inference problem as a Mealy machine inference problem results in more compact models and provides a simple, broadly understood formal underpinnings for our work. Second, we propose a highly effective prediction technique for minimizing the number of queries generated during the inference process. Third, we propose two optimizations to the basic $L*$: parallelization and caching.

On the analysis side, we show how the computed complete models can be used as a formal basis to study and defeat botnets: First, we show how to identify the weakest links in a protocol, especially in the context where multiple pools of bots partially share the same resources. Such weakest links are critical for normal functioning of one or more agents participating in the protocol. Second, we show that the inferred model can be used to uncover protocol design flaws. Third,

we demonstrate how inferred models can be used to prove the existence of unobservable communication back-channels among botnet servers, although we have no access to those communication channels. Besides proving the existence of such channels, the analysis we propose can actually construct a state-machine representing the model of the back-channel communication. Fourth, we demonstrate how complete models can be used for detecting differences between distinctive implementations of the same protocol.

On the experimental side, we show how to design an effective protocol inference system and provide empirical evidence that our optimizations — query response prediction, parallelization, and caching — speed up the inference process by over an order of magnitude compared to the basic $L*$ algorithm. For instance, our prediction technique alone reduces the time required for the MegaD model inference from an estimated 4.46 days to 12 hours. Applying formal analysis to inferred complete models, we uncover previously unknown facts about the MegaD botnet. Analyzing critical links, we identify the critical components of the botnet shared among multiple pools of bots. Analyzing the properties of the inferred model, we discover a design flaw in MegaD. More precisely, we find a way to bypass MegaD's master server authorization and gain unlimited access to fresh spam templates, which can be used to train spam filters even before a significant percentage of bots starts sending spam based on those templates. Analyzing the back-channels, we prove that MegaD's servers communicate with each other, and we even construct a formal model of such communication. Analyzing differences among Postfix and MegaD's SMTP implementations, we discover a number of interesting differences, useful for detection and fingerprinting. We validate our technique by inferring the complete protocol state-machine from Postfix's SMTP implementation, checking its equivalence against the standard.

3.2 Problem Definition

A communication protocol is a set of rules for exchanging information over some medium (e.g., the Internet). These rules regulate data representation (i.e., the message format), encryption, and the state-machine of the communication. Any automatic technique for reverse-engineering of real-world protocols has to deduce message formats, handle encryption, and infer state-machines. The first two components of the problem have received significant attention of the research community [13, 15, 29, 30, 92]. The third component — protocol state-machine inference — has received far less attention, and is the focus of our work.

In the first part of this section, we define the model inference problem informally, and leave a more formal treatment for the subsequent sections. In the second part of this section, we go further to propose several automatic analyses of the model. We define several related problems, which can be solved precisely only if a complete state-machine of the protocol is known. In the third part of this section, we outline our assumptions.

Model Inference

The goal of protocol model inference is to learn a state-machine describing the protocol composed of a finite set of states and a transition relation over a finite alphabet. In general, it is not possible to learn a completely accurate model, without having access to a source of counterexamples that show when the learned model differs from the actual system [4]. Thus, every protocol inference approach is necessarily an ε-approximation, i.e., the inference cost is proportional to the desired accuracy ε and confidence γ.[1] When counterexamples are available, $L*$ can make at most a polynomial number of queries, but in the approximation setting, the number of queries depends on the desired accuracy and confidence.

There are two basic types of finite state-machines: the Moore machine [69] and the Mealy machine [67]. Informally, the former distinguishes states according to whether they are accepting or not, while the latter has no accepting states and distinguishes states according to the sequence of outputs produced from a sequence of transitions. Since protocols are reactive systems[2], the Mealy machine is a more appropriate model. The problem this chapter addresses is how to learn the Mealy machine describing the studied protocol in the realistic network setting, treating the protocol implementation as a black-box and learning the state-machine from active probing. We actually set the bar higher: The problem we want to solve is to learn the minimal (the fewest states) complete (transitions defined for all inputs and states) Mealy machine describing the protocol.

Model Analysis

Once the protocol model is constructed, we wish to perform four types of analysis: identification of the critical links in the protocol, identification of protocol design flaws, proving the existence of the background communication over unobservable channels, and (dis)proving equivalence of different implementations of the same protocol (a.k.a. equivalence checking [55]).

Identification of the critical links in a protocol is important for optimizing the attacks on the botnet. We define the problem as follows: Given an initial state of the protocol and some set of bad actions (e.g., spamming), represented with output responses,[3] that we want to prevent from happening, what is the minimal set of transition edges that need to be disrupted in order to prevent the bots from executing those actions? The bad actions can be disrupted either by making it impossible for the bots to reach the state from which the bad action is executed, or by disrupting the bad actions themselves.

Identification of protocol design flaws can be done through manual inspection of the model or automatic model checking (e.g., [25]). In either case, the analyst needs to come up with a set of properties and then check whether the model satisfies them. For instance, one of the properties we checked was: "Bot cannot obtain spam templates before (1) being authenticated by the master server and (2) getting a command to download spam templates."

[1] The ε accuracy should not be confused with the empty string, also denoted ε.

[2] Reactive systems maintain an ongoing interaction with their environment rather than produce some final value upon termination.

[3] Bad output responses in a Mealy machine correspond to bad states of an equivalent Moore machine.

Proving the existence of background communication among servers whose communication we cannot eavesdrop is important for gaining knowledge about the communication over channels we have no access to. The knowledge of existence of such channels can help security researchers in detecting infiltration traps. We define the problem as follows: Given a client (a bot) communicating with a certain number of servers (three in the MegaD case: the master, SMTP [53], and template server), can we prove existence and build a model of inter-server communication, only from the observed communication between the client and servers?

Equivalence checking (e.g., [55]) is an automatic analysis that takes two formal models and either proves that models are equivalent, or finds counterexamples showing the differences. In our setting, we were especially interested in finding differences between MegaD's custom SMTP implementation and the standard SMTP to detect the features that could be used for fingerprinting. In a broader setting, equivalence checking can be used for detecting deviations from the standard, differences among different implementations of the same protocol, and uncovering implementation flaws.

Assumptions

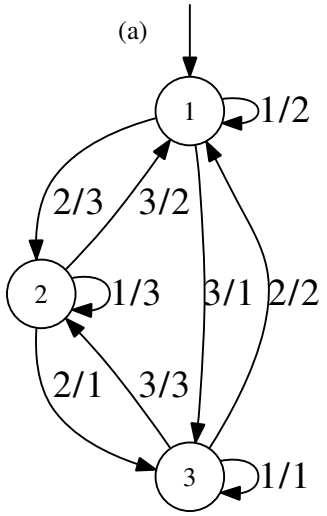
Determinism. We assume that the protocol to be learned is deterministic, i.e., that the same sequence of inputs from the initial state always produces the same sequence of outputs and ends in the same state. Among all the protocols we studied, we found only one minor easy-to-handle source of non-determinism in the MegaD botnet protocol. We explain later how we handle that specific source of non-determinism. Some limited amount of non-determinism can be handled by extending the alphabet. For example, if a protocol implementation in state s responds to message m either immediately with r or waits for ten seconds and responds with t, one can split m into two messages, m_0 and m_{10}, such that the response to m_0 (resp. m_{10}) is r (resp. t). Each of the two messages can transition from s into different states.

Resettability. State-machine inference algorithms require the means of resetting the machine to a fixed start state. The sequence of inputs that reset the state-machine into its start state is known as a *homing sequence* and every finite state-machine has such a sequence [79]. Network protocol state-machines are often easily reset by initiating a new connection or session. Thanks to the prior work on the message format reverse-engineering [14, 29, 30], we know the messages that reset the state-machines of the protocols we study.

Finiteness. We are interested in inferring finite-state protocols (or finite abstractions of infinite-state ones) with finite input and output alphabets. We leave the inference of more expressive models for the future work.

Known Message Format Semantics. To abstract input and output messages with a finite alphabet, we rely upon the previous work on automatic message format reverse-engineering [15]. Another interesting option is the automatic message clustering and abstraction proposed by Comparetti et al. [27].

Known Encryption. Many botnet protocols encrypt messages, complicating protocol inference. If protocol messages are encrypted, our approach needs to know the protocol encryption and decryption functions. We leverage the previous work of Caballero et al. [13] to automatically extract



(b)

		E		
		1	2	3
S	ε	2	3	1
	2	3	1	2
	3	1	2	3
$S \cdot \Sigma_I$	1	2	3	1
	$2 \cdot 1$	3	1	2
	$2 \cdot 2$	1	2	3
	$2 \cdot 3$	2	3	1
	$3 \cdot 1$	1	2	3
	$3 \cdot 2$	2	3	1
	$3 \cdot 3$	3	1	2

Figure 3.2: A Mealy Machine and the Corresponding Observation Table. The initial state is denoted by an incoming edge with no source. Edges are labeled with input/output symbols. In this thesis, all three set of labels — state, input, and output labels — are unrelated sets of integers.

and circumvent encryption functions.

3.3 Background Material

This section briefly surveys prior work required for understanding this chapter. In the first half, we provide a brief overview of the main definitions and results in the automata theory and inference. In the second half, we briefly describe botnets, which are our main targets. Even though the targets that we study are highly specific, the results of our research are applicable to a broad range of protocols in the realistic network setting.

Inference of Mealy Machines

Protocol inference is a special case of the grammatical inference problem [45], because the problem of the finite state-machine inference is isomorphic to the problem of learning a regular language. In this thesis, we solely use Mealy machines, as they are a more appropriate model for reactive systems, which have neither accepting nor rejecting states, and are more compact than Moore machines [69] by a linear factor equal to the size of the output alphabet.

Definition 1 (Mealy Machine [67])**.** *A Mealy machine is a six-tuple* $(Q, \Sigma_I, \Sigma_O, \phi, \lambda, q_0)$*, where* Q *is a finite non-empty set of states,* $q_0 \in Q$ *is the initial state,* Σ_I *is a finite set of input symbols,* Σ_O *is a finite set of output symbols,* $\phi : Q \times \Sigma_I \longrightarrow Q$ *is the transition relation, and* $\lambda : Q \times \Sigma_I \longrightarrow \Sigma_O$ *is the output relation.*

In this thesis, we focus on inferring *complete* protocol state machines. A complete Mealy Machine is defined as follows:

Definition 2 (Complete Mealy Machine)**.**
A Mealy machine $(Q, \Sigma_I, \Sigma_O, \phi, \lambda, q_0)$ *is complete if and only if* ϕ *and* λ *are defined* $\forall m \in \Sigma_I$ *and* $\forall q \in Q$.

We rely upon prior work [14] for reverse-engineering of the alphabet and claim no contribution on that front. We are not aware of any automatic technique capable of reverse-engineering the complete alphabet, so we necessarily have to work with a subset. Given a subset of the alphabet, our technique infers a complete state-machine (Definition 2), in contrast to prior work on protocol inference that learns incomplete state-machines [27, 48]. To the best of our knowledge, our work is the first to demonstrate a technique for complete protocol state-machine inference in the realistic network setting.

In this thesis, input symbols will be denoted with letters from the beginning of the alphabet a, b, c, output symbols with letters from the end of the alphabet x, y, and strings of input symbols with letters r, s, t, u. The set of all symbols in a given string is denoted as $[\,]$, e.g., $[a \cdot b \cdot b] = \{a, b\}$, where '$\cdot$' is the concatenation operator. We extend the ϕ and λ relations to strings of characters from Σ_I, for example, $\phi^*(q, a \cdot b \cdot c) = \phi(\phi(\phi(q, a), b), c)$ and $\lambda^*(q, a \cdot b \cdot c) = \lambda(q, a) \cdot \lambda(\phi^*(q, a), b) \cdot \lambda(\phi^*(q, a \cdot b), c)$. We shall frequently use a more intuitive term *response* for a string of output symbols returned by λ^*.

Angluin [4] proposed the first polynomial algorithm, called $L*$, for learning Moore machines. Shahbaz and Groz [82] adapted the algorithm for Mealy machines and proposed several optimizations. In this section, we briefly describe Shahbaz and Groz's algorithm, while in the following section, we describe our improvements and optimizations of their algorithm. The $L*$ algorithm maintains an observation table:

Definition 3 (Observation Table [4])**.** *An observation table is a triple* (S, E, T)*, where S is a prefix-closed*[4] *set of strings of input symbols from* Σ_I^* *and set E is a suffix-closed set of strings from* Σ_I^+*. Let set* $S \cdot \Sigma_I$ *be defined as a concatenation of all strings in S with every alphabet symbol in* Σ_I*, i.e.:* $S \cdot \Sigma_I = \{s \cdot a \mid s \in S, a \in \Sigma_I\}$*. Then map* $T : (S \cup S \cdot \Sigma_I) \times E \longrightarrow \Sigma_O^+$ *is a map from* $S \cup S \cdot \Sigma_I$ *and E to a non-empty sequence of output symbols. Map T has an additional property:* $\forall s \in S \cup S \cdot \Sigma_I, r \in E, x \in \Sigma_O^+ \,.\, (T(s, e) = x) \Rightarrow |r| = |x|$.

Intuitively, S and $S \cdot \Sigma_I$ can be seen as two sets of rows, E as a set of columns, and T as a map that maps input strings obtained by concatenation of rows $(S \cup S \cdot \Sigma_I)$ and columns (E) to output strings of the same length as the column part of the input string. For example, from Table 3.2b, it follows that $T(2 \cdot 1, 3) = 2$, where $2 \cdot 1 \in S \cdot \Sigma_I$ is a row and $3 \in E$ is a column.

Angluin introduced two observation table properties: closure and consistency. Only the first property is important for understanding this thesis, while the second is always satisfied if T is computed in a specific way [82].

[4]Let *Pref* be a function that returns a set of all prefixes of a string. Set S is prefix-closed if and only if $\forall s \in S \,.\, Pref(s) \subseteq S$. Note that an empty string ε is a prefix of every string. The definition of a suffix-closed set is similar.

Algorithm 1: The $L*$ Algorithm: Closing the Observation Table.

$S := 0$;
$S \cdot \Sigma_I := \{\varepsilon\}$;
$E := \Sigma_I$;
while $\exists s \in S \cdot \Sigma_I \,.\, (\forall t \in S \cdot \Sigma_I \,.\, s \leq t) \wedge (\forall t \in S \,.\, \exists r \in E \,.\, T(s,r) \neq T(t,r))$ **do**
    Move s to S;
    forall the $a \in \Sigma_I$ **do**
        $S \cdot \Sigma_I = S \cdot \Sigma_I \cup s \cdot a$;
        forall the $u \in E$ **do**
            Compute response to $s \cdot a \cdot u$;
            Take suffix x of the response (last $|u|$ symbols);
            Update table $T(s \cdot a, u) = x$;
Run sampling and Algorithm 2 if there is a counterexample;

Definition 4 (Closed Observation Table [4])**.** *We say that an observation table is* closed *if and only if* $\forall s \in S \cdot \Sigma_I \,.\, \exists t \in S \,.\, \forall r \in E \,.\, T(s,r) = T(t,r)$.

Intuitively, a table is closed if for every row s in the $S \cdot \Sigma_I$ set of rows, there is a row t in S having exactly the same responses, i.e., $\forall r \in E \,.\, T(s,r) = T(t,r)$. For example, Table 3.2b is closed. Table T is usually computed in such a way that no two rows in S are equivalent, i.e., there is always only one representative in S of each class of equivalent rows. For example, rows ε and $2 \cdot 3$ in Table 3.2b are equivalent, so only one representative (ε) is in S. Further, the representative put in S is minimal, according to the following ordering.

Definition 5 (Lexicographic Ordering)**.** *Let lexicographic ordering relation* $<: \Sigma_I^* \times \Sigma_I^* \longrightarrow Bool$ *be a total order relation over two strings of input symbols, say* $s = a_0 \cdots a_n$ *and* $t = b_0 \cdots b_m$*, defined as follows.*

$$s < t = \begin{cases} |s| < |t| & \text{if} \quad |s| \neq |t| \\ a_j <_a b_j & \text{if} \quad s \neq \varepsilon \wedge a_j \neq b_j \wedge \forall 0 \leq i < j < |s| . a_i = b_i \\ \mathit{false} & \mathit{otherwise} \end{cases}$$

Define $s \leq t$ *as* $s < t \vee s = t$*. The alphabet symbols are ordered according to some arbitrary total ordering* $<_a$*.*

Intuitively, the definition imposes a total ordering on strings, given a total order of the alphabet symbols, e.g., if $a <_a b <_a c$, then $b \cdot a < b \cdot c$. We use this ordering in our implementation of the $L*$ algorithm (Algorithm 1) to choose the minimal representative of a class of equivalent rows.

Next, we give a high-level description of the $L*$ algorithm. $L*$ has two steps. In the first step, $L*$ initializes S to a set containing an empty string, $S \cdot \Sigma_I$ to an empty string ε, the columns E to the input alphabet[5], and T to the responses of the system under study to inputs constructed

[5]The algorithm can also trivially handle monotonically increasing alphabets, useful when new messages are discovered during the inference, but we abstracted that away for clarity.

from concatenating the rows (i.e., an empty string at this point) with columns (individual alphabet symbols at this point). The algorithm then iteratively keeps closing the table and re-computing the responses to all $S \cdot \Sigma_I \cdot E$ sequences of messages. Each sequence has to be generated, transmitted on the network, response recorded, and stored into table T. The $S \cdot \Sigma_I \cdot E$ queries generated in the first step of $L*$ are called membership queries.

In the second step, $L*$ constructs a complete minimal conjecture automaton from a closed table. This construction is performed as follows: The rows in S represent the minimal set of states, rows in $S \cdot \Sigma_I$ represent transitions for every symbol in Σ_I, and the range of T (i.e., the elements of the table) represent the output relation. The conjectured automaton is then, in the original algorithm [4], passed to a *teacher*, which either confirms that the machine has been correctly learned, or returns a counterexample. The counterexample is a sequence of inputs and corresponding outputs that does not match the conjectured machine. Of course, we cannot ask the bot master to tell us whether our conjectured state-machine is correct or not, so we use a sampling-based approach [4], which randomly generates uniformly distributed sequences of messages used to sample the protocol's state-machine and discover mismatches. If the response of the agents involved in the message exchange does not match those predicted by the conjectured state-machine, we have a counterexample. Based on the number of sampling sequences and the number of conjectures, one can compute the accuracy and confidence that the conjectured machine is correct, as proposed by Angluin [4]. The sampling queries generated in the second step of $L*$ are called equivalence queries, for they check the equivalence between the conjectured and to-be-learned machine.

Algorithm 2: The $L*$ Algorithm: Handling Counterexamples.

```
Let r,x be a counterexample input (r) and its response (x)
Let r = p·s, s.t. p ∈ S∪S·Σ_I and p is the longest such prefix
forall the u suffixes of s do
    E = E∪u forall the t ∈ S∪S·Σ_I do
        Compute response to t·u;
        //If t·u = r, response is x
        Take suffix y of the response (last |u| symbols);
        T(t,u) = y;
Run the inner loop of Algorithm 1 if T changed;
```

Algorithm 2 illustrates Shahbaz and Groz's algorithm for handling counterexamples. When a counterexample, an input (r) and the corresponding response (x), is found, the algorithm finds the longest prefix of r existing in $S \cup S \cdot \Sigma_I$, and trims that prefix. The remaining suffix and all its suffixes are added to the columns (set E) of the observation table, and the responses for all the rows extended with newly added suffixes are computed.

The number of required membership queries depends on the size of the state-machine to be inferred. Even a twenty-state machine could require tens of thousands of queries. To reduce the number of membership queries, we developed a novel response prediction heuristic, which exploits redundancy in inferred models. Any mispredictions are guaranteed to be detected by the

sampling queries with the desired accuracy and confidence. Since the number of sampling queries is computed solely from the accuracy, confidence, and the number of conjectures [4], detecting mispredictions does not require any additional sampling queries.

Our response prediction heuristic exploits the abundance of self-loop transitions, defined as follows.

Definition 6 (Self-Loop Transitions)**.** *Let q be some state and a some input symbol. Transitions for which $\phi(q,a) = q$ are called* self-loop transitions*.*

Botnets

A botnet is a network of compromised hosts controlled remotely by botmasters to carry out nefarious activities such as denial of service, theft of personal information, and spamming. Botmasters control their botnets through a system of Command and Control (C&C) servers. MegaD is a mass spamming botnet first observed 2007, and was responsible for one-third of the world's spam at its peak[6]. The main MegaD C&C server used by each pool of bots is the Master server, which points bots to a set of auxiliary (Template, SMTP, and Drop) servers. A spamming MegaD bot contacts only the Template and SMTP servers [21].

MegaD has been the target of multiple takedown attempts. The most recent attempt occurred in Nov. 2009, but MegaD bounced back after the takedown. The common practice to botnet takedowns is to identify as many C&C servers as possible, and attempt to cripple the entire botnet by sending abuse notifications to all ISPs involved simultaneously.[7] This is an expensive exercise requiring careful co-ordination among all parties involved. In this chapter, we show through protocol inference that MegaD's SMTP servers are actually the critical link in the C&C, and taking just SMTP servers down would be a cheaper and simpler option.

3.4 Inference of Protocol Models

This section presents our technique for inference of complete protocol state-machines in the realistic network setting. The high-level architecture of our implementation is shown in Figure 3.3. Our implementation is composed of several components: a bot emulator, a query cache, a membership query predictor, and $L*$.

The bot emulator is a script we wrote that receives queries (strings of symbols from the input alphabet) from $L*$ and concretizes them into valid protocol messages sent to botnet servers. Once the bot emulator receives a response, it abstracts the response into strings of symbols from the output alphabet and sends such abstracted strings to $L*$. We describe abstraction and concretization in Section 3.4.

We built our bot emulator from scratch, to assure it cannot perform any of the malicious activities (spamming and infection) of the real bot. In addition, we carefully crafted our experiments not

[6] http://www.m86security.com/trace/i/Mega-D,spambot.896∼.asp

[7] http://blog.fireeye.com/research/2009/11/smashing-the-ozdok.html

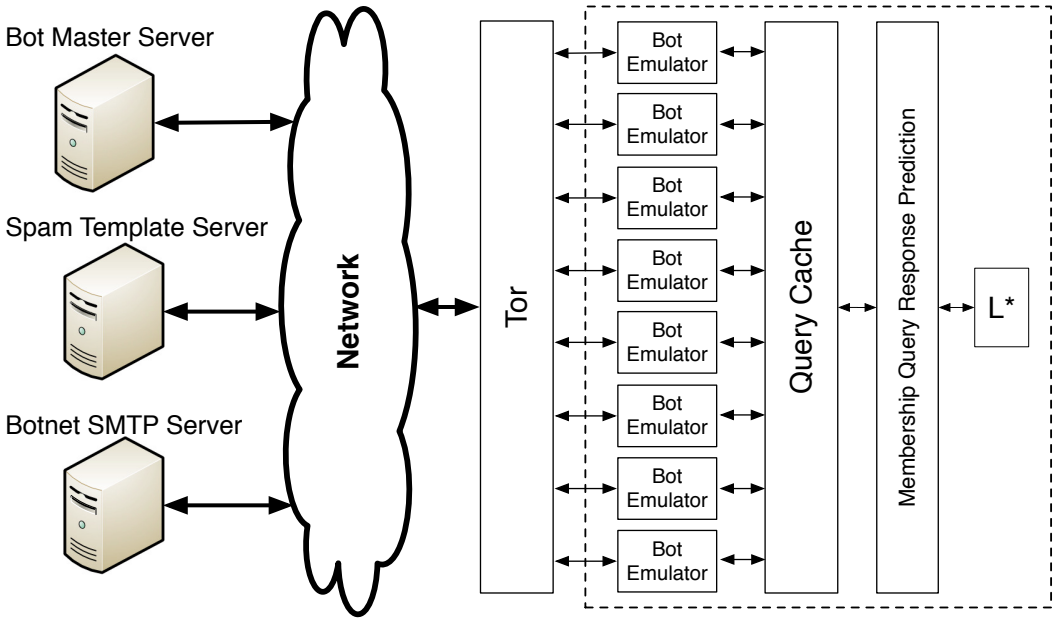


Figure 3.3: Architecture of Our Protocol Inference Engine. Our inference system is encircled with a dashed line.

to cause any harm to any party involved (infected users, ISPs, C&C servers, and botmasters). For example, our bot emulator is careful not to construct corrupted messages intentionally, minimizing the impact even on the C&C servers.

The query cache acts as a concentrator of parallel query responses and caches the results, so that each sequence of messages has to be sent over the network only once. Through parallelization, we achieved 4.85X reduction in the time required for the inference using eight machines, each running one bot emulator. We describe both parallelization and caching in Section 3.4.

The membership query predictor attempts to predict what is the most likely response to membership queries. Learning even a state-machine of a medium size can require a significant number of queries (c.f. Section 3.3). As queries can be long strings of input messages, and getting a response to each message can take a long time due to network delay (6.8 sec on average in our experiments), accurate prediction of responses is important to infer protocol state-machines in reasonable amount of time. Erroneous predictions are guaranteed to be detected (with desired accuracy and confidence) using sampling queries and fixed by backtracking to the first mistake made by the predictor. We present our prediction heuristic in Section 3.4.

In Section 3.4, we explain how we handled the only discovered source of non-determinism in the MegaD protocol, state-machine resetting, and generation of sampling queries.

Message Abstraction and Concretization

$L*$ constructs queries over the abstract input alphabet and passes them to our bot emulator, which concretizes the alphabet symbols into valid network messages and sends them to botnet servers. When responses are received, our emulator does the opposite — it abstracts the response messages into the output alphabet and passes them to $L*$. Construction of the input and output alphabets is a partially automatic and partially manual process. We reverse-engineer the message formats and their semantic content using automatic protocol reverse engineering [14] and encryption/decryption modules extracted from the bot binary [13]. Once we learn the message formats, we perform abstraction manually. The manual abstraction is a straight-forward process, but requires intelligence in deciding which message fields are important. In particular, the most important field, not surprisingly, turns out to be the message type field.

Besides abstraction, another important aspect of automatic protocol inference is concretization of messages, i.e., generation of valid network messages. If messages are invalid or have invalid session tokens, the server will simply reject them. To generate valid messages, the bot emulator uses the automatically reverse-engineered message format grammar, rewrites the message fields as needed, and encrypts the messages before transmission. The bot emulator rewrites the tokens of server-bound messages using tokens issued by the server in the same session. If the token has not been issued, the emulator rewrites tokens with a random value, to learn how servers handle invalid tokens.

To assure our experiments are reproducible, we include MegaD's C&C message format tree in Figure 3.4 and a list of all abstracted messages used in this work in Table 3.1. MegaD uses a proprietary C&C protocol for communication with its master and template servers, and a non-standard SMTP protocol for communication with its SMTP server. To model the C&C protocol messages, we use three fields: the *MsgType* field found in all messages, the *SubType* field in the INIT and GETCMD messages, and the *Config* element found only in the spam template messages. We define a unique symbol for each unique $\{$*MsgType*, *SubType*, *Config*$\}$ combination as described in Table 3.1. To model the bot's communication with the SMTP server, we abstract MegaD's SMTP dialogs at two different levels of abstraction. When we model the role of the SMTP server in the overall C&C protocol, we abstract MegaD's spam capability test dialog and pre-spam notification dialog each with two symbols, one in either direction (IDs 14 and 15 in upper half and IDs 12 and 13 in lower half of Table 3.1 respectively). When we analyze the details of MegaD's SMTP protocol implementation, we abstract each individual SMTP message within a TCP connection into individual symbols.

Query Cache

In our implementation, the query cache is just a file that stores pairs of input message sequences and the corresponding responses (i.e., sequences of output messages). The cache acts primarily as a concentrator of parallel query responses. This architecture (Figure 3.3) simplifies the implementation of $L*$, for only the queries have to be issued in parallel, but responses can be processed sequentially. More precisely, the membership query loop starting on Line 6 of Algorithm 1 and the

ID	MsgTy	SubTy	Template	Semantic Label	Direct.
-	0x00	0x00	-	INIT	→ MS
1	0x00	0x01	-	GETCMD	→ MS
2	0x03	-	-	DLSUCCESS	→ MS
3	0x05	-	-	DLERROR	→ MS
4	0x06	-	-	DL1	→ MS
5	0x09	-	-	DL2	→ MS
6	0x16	-	-	HOSTINFO	→ MS
7	0x22	-	-	CAP_TESTPASS	→ MS
8	0x23	-	-	CAP_TESTFAIL	→ MS
9	0x25	-	-	TS_RECVED	→ MS
10	0x09	-	-	PONG	→ MS
11	0x1c	-	-	TEMPLATE_ACK	→ TS
12	0x1d	-	-	GET_TEMPLATE	→ TS
13	0x1e	-	-	SPAM_STATUS	→ TS
14	-	-	-	TEST	→ SS
15	-	-	-	NOTIFY	→ SS
-	0x01	-	-	INFO	← MS
1	-	-	-	TCP_CLOSE(-)	← MS ← TS
2	0x04	-	-	ACK_DLRESULT	← MS
3	0x07	-	-	ACK_DL1	← MS
4	0x0a	-	-	ACK_DL2	← MS
5	0x0e	-	-	WAIT2	← MS
6	0x15	-	-	GETHOSTINFO	← MS
7	0x18	-	-	WAIT1	← MS
8	0x1c	-	None	TEMPLATE	← TS
9	0x21	-	-	TESTSPAMCAP	← MS
10	0x24	-	-	DLTEMPLATE	← MS
11	0x1c	-	RENEW	RENEW	← TS
12	-	-	-	TESTPASS	← SS
13	-	-	-	NOTIFY_RECVED	← SS

Table 3.1: Abstraction of MegaD's Communication with its Master and Template Servers. The upper half of the table shows the input alphabet (sent by our bot emulator to various servers), and the lower half the output alphabet (responses sent by servers to our bot). We use the MsgType, SubType and Template config fields for abstraction. Messages sent to and received from C&C servers are abstracted as input and output alphabet symbols (the ID column) respectively, where MS is the Master Server, TS is the Template Server and SS is the SMTP Server. The no-response message is denoted as TCP_CLOSE(-). The INIT message is sent to reset each session. The INFO message is always followed by another message, and that second message is used for abstraction. The SMTP messages are abstracted according to the standard [53].

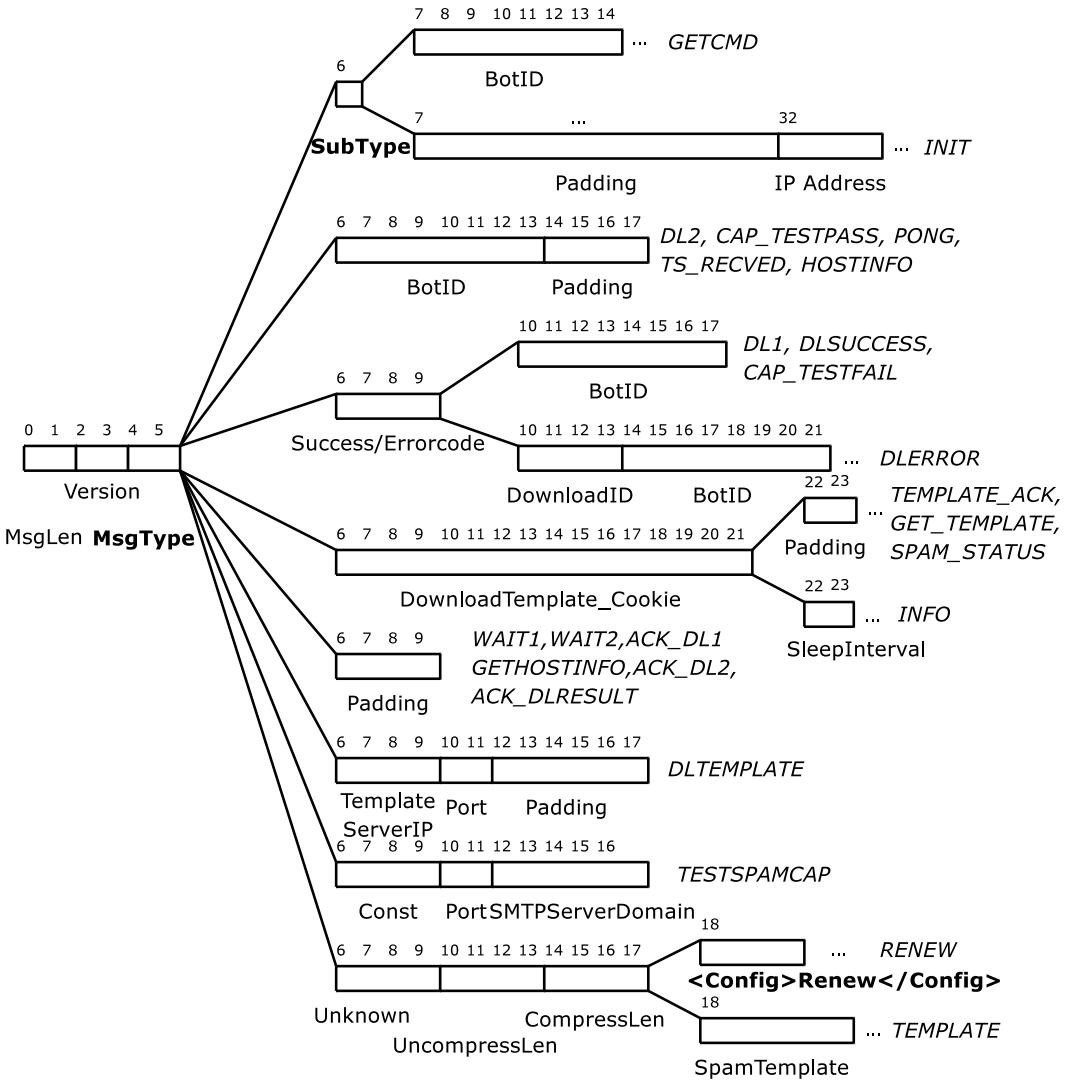


Figure 3.4: MegaD's C&C Message Format Tree. Message type fields used for abstraction are shown in bold. We elide the tails of messages not relevant to our abstraction process. Abstracted messages are shown on the right in italics.

column extension loop starting on Line 3 of Algorithm 2 both execute a number of independent membership queries, which can all be either predicted or run on the network in parallel. Our implementation of $L*$ emits all these queries in parallel. The queries are partitioned among a number of machines (eight, in our case) and independently run on the network. In addition, the cache has a more traditional role of caching the responses that have been tested on the network — such responses can be reused at any point, because of the determinism assumption (Section 3.2).

Response Prediction

Studying the membership queries that $L*$ makes while learning protocol models, we came to realize that there is a significant amount of redundancy in the inference process. Namely, many states have many self-loops (Definition 6). The figure on the left illustrates this phenomenon on a small piece of the MegaD state-machine. Such self-loops increase the number of membership queries, without helping $L*$ to distinguish states. We believe that there are two major factors contributing to the abundance of self-loops: (1) Our goal is to learn complete models, which means we need to determine the response to *all* input alphabet symbols (i.e., messages) from every state. Most protocols use each message for a single, well-defined purpose, which means that most often sending an unexpected message from some state causes that message to be ignored, either at that state or in an error state. (2) We need to abstract conservatively the messages into the input and output alphabets *before* the protocol state-machine is known. The conservative overestimation frequently increases the redundancy in the model, e.g., increasing the number of self-loops, which cannot be eliminated before the state-machine is known — a chicken and egg problem. On the other hand, reducing the size of the alphabet oversimplifies the inferred state-machine. Thus, it is important to be conservative in choosing the alphabet and develop effective techniques for improving the performance of the inference process.

To understand the intuition behind response prediction, consider the running example shown in Figure 3.2a. There are three self-loops, and we shall focus on the self-loop incident to state 2. Since self-loops return back to the same state, the response to $2 \cdot 1 \cdot u$ (with self-loop) and $2 \cdot u$ (without self-loop) will be the same for every input string u. Accordingly, the table entries for $2 \cdot 1 \cdot u$ (row 5) and $2 \cdot u$ (row 2) in Table 3.2b are the same. Recall from Algorithm 1 that the length of membership queries increases with each outer-for-loop iteration. Thus, the $2 \cdot u$ query must preceede the $2 \cdot 1 \cdot u$ query, and the response to the former can be used to predict the response to the latter at the end of each outer-for-loop iteration.

We exploit the redundancy with a two-level heuristic. The first level, called *restriction-based* prediction, exploits the abundance of self-loops, for guessing responses to membership queries with high accuracy. The second level, called *probability-based* prediction, exploits the fact that the same input messages often leads to the same output message. Any possible prediction errors are detected using random sampling (Section 3.3). If an error was made predicting the responses to membership queries, our algorithm backtracks to the first erroneously predicted membership query, fixes the error (according to responses from sampling queries), and continues running $L*$. Importantly, the same sampling queries can be used both to detect missing states (as in the classical Angluin probabilistic sampling [4]) and mispredictions.

We begin the formal presentation of our prediction techniques by showing that entries in the S set have no self-loops, and therefore can be used for prediction of responses that have self-loops.

Theorem 1. *Let $s \in S, s = c_0 \cdot c_1 \cdots c_n$ be a string of n characters from Σ_I. Let $q_0, q_1, \cdots, q_n, q_{n+1}$ be a sequence of states visited by $\phi^*(q_0, s)$. For $0 \leq i < n+1$, every two adjacent states are different, i.e., $q_i \neq q_{i+1}$.*

Proof. Proof is by induction on the string length. The ε string trivially satisfies the condition, as there is only one state in the sequence of visited states. Let $s = c_0 \cdots c_k$ and $s \in S$. Algorithm 1 on Line 1 appends $a \in \Sigma_I$ to s, and then in the loop that follows computes responses to membership queries $s \cdot a \cdot u$, where $u \in E$ are the columns of T. Without loss of generality, let $s \cdot a$ be the smallest string, according to our lexicographic ordering (Definition 5). String $s \cdot a$ is moved to S (Line 1) only if $\forall t \in S \;.\; \exists u \in E \;.\; T(t,u) \neq T(s \cdot a, u)$. Every $s \cdot a$ string is obtained by extending strings from S, so we know that $s \in S$. Thus, we conclude: $\exists u \in E \;.\; T(s,u) \neq T(s \cdot a, u)$, which implies that $\phi^*(q_0, s \cdot u) \neq \phi^*(q_0, s \cdot a \cdot u)$, because s and $s \cdot a$ are (eventually) both in S and they denote different states. Since $s \cdot a \cdot u$ and $s \cdot u$ have the same prefix and suffix, symbol a must explain the difference in the response of the state-machine. By induction hypothesis $\forall 0 \leq i < k \;.\; c_i \neq c_{i+1}$. So we have: $\phi^*(q_0, s \cdot a) = c_0 \cdots c_k \cdot c_{k+1}$. Assume $c_k = c_{k+1}$. Thus, $\phi^*(q_0, s \cdot a) = \phi^*(q_0, s)$, a contradiction. So, we have proved that in $\phi^*(q_0, s \cdot a)$, the last two visited states are different, which proves the theorem. □

Corollary 1. *From Definition 6, it follows that $\phi^*(q_0, s)$ for $s \in S$ has no self-loop transitions.*

Intuitively, Theorem 1 suggests that strings from S, which are loop-free, could be used to predict responses to input strings that are similar to strings from S, but have additional symbols producing self-loop transitions sprinkled around. We formalize that intuition using the following two definitions.

Definition 7 (Differentiating Set)**.** *Let $D \subseteq \Sigma_I$ be a set of all symbols in S, more precisely, $D = \{c \mid c \in [s], s \in S\}$. As S grows during the execution of Algorithm 1, D is going to be a monotonically increasing set.*

Definition 8 (Restriction Function)**.** *Let $a \in \Sigma_I$, $s \in \Sigma_I^*$, and $D \subseteq \Sigma_I$. The restriction function $\rho : \Sigma_I^* \times D \longrightarrow \Sigma_I^*$ is then defined recursively as follows:*

$$\rho(s, D) = \begin{cases} \varepsilon & \text{if } s = \varepsilon \\ \rho(r, D) & \text{if } s = a \cdot r \;\wedge\; r \in \Sigma_I^* \;\wedge\; a \notin D \\ a \cdot \rho(r, D) & \text{if } s = a \cdot r \;\wedge\; r \in \Sigma_I^* \;\wedge\; a \in D \end{cases}$$

Intuitively, the restriction function just deletes the symbols that are not in a given set from a string. For example, $\rho(a \cdot b \cdot b \cdot a \cdot c \cdot d, \{b, d\}) = b \cdot b \cdot d$.

Now, our restriction-based prediction rule can be simply stated as: Given any membership query $s \cdot a \cdot u$, compute $s' = \rho(s \cdot a, D)$, and if s' already exists in $S \cup S \cdot \Sigma_I$, use the values in the s' row of table T to predict the values for the $s \cdot a \cdot u$ row. More formally: $T(s \cdot a, u) = T(\rho(s \cdot a, D), u)$, if

the $\rho(s \cdot a, D)$ entry exists in T. With this simple rule, we get highly accurate prediction, with few errors.

The restriction-based prediction saves around 73% of membership queries in our experiments with MegaD. Analyzing the results, we identified one missed prediction opportunity. If the restricted input string does not exist in the table, the previously presented technique is helpless. To improve the performance of the restriction-based prediction even further, we track the set of observed response messages for every input symbol. When the restriction-based approach fails, we apply a simple probability-based prediction: If a particular input message produces the same output message for *all* previous queries, we predict that response will be the same. If multiple different responses were observed for the same input, we do not predict it. It would certainly be possible to lower the prediction threshold — say, by picking the response that happens in at least 90% of cases — at the cost of increasing the number of erroneous predictions and the cost of backtracking. We leave this fine-tuning for future work. Even with the simple probability-based prediction currently implemented, we gain an additional 13% reduction in the number of membership queries on MegaD, in addition to savings achieved by the restriction-based prediction.

Mispredictions can produce erroneous state-machine conjectures. However, we exploit the same random sampling equivalence checking mechanism in $L*$ (Section 3.3) to detect mispredictions. Thus, mispredictions are guaranteed to be found with desired accuracy ε and confidence γ. Once an error is detected, $L*$ backtracks to the first erroneously predicted query, fixes it using the sampling query response, removes all subsequently predicted entries from the observation table, and continues the inference process. All the prediction savings, both in the prior discussion and in the experimental evaluation, take the cost of backtracking into account. So, it is obvious that our prediction is very effective (86% total reduction in the number of queries) and accurate (inaccurate prediction would require more frequent backtracking, reducing the savings).

Determinism, Resettability, and Sampling

This section describes the non-standard and non-obvious aspects of using $L*$ in our setting. More precisely, we discuss the impact of the determinism and resettability assumptions (Section 3.2) and the role of the sampling process in achieving the desired accuracy of the model.

Both the determinism and resettability assumptions were relatively easy to satisfy in our setting. In our experiments, the exchange of messages was deterministic, except for one corner case: Sometimes master servers respond with an arbitrarily long sequence of INFO messages, which are always terminated with a non-INFO message. Our inference infrastructure discards all the INFO messages, and treats the first non-INFO message as the response. This was the only source of nondeterminism we encountered. To reset the state-machine, we begin both membership and sampling queries with an INIT message (c.f. Table 3.1), which initiates a new session. Once the session is started, every input message produces a response — an output message.

As discussed in Section 3.3, we use a sampling-based approach for equivalence queries. We generate uniformly distributed random sequences of input messages, the number of which is determined by the desired model accuracy and confidence [4]. Once our implementation of $L*$ closes the table, it conjectures a state-machine, which is then tested through sampling. The responses to

sampling queries are never predicted, for the purpose of sampling queries is to discover new states that do not exist in the currently conjectured model and to discover prediction errors.

3.5 Analysis of Inferred Models

In this section, we analyze the complete protocol models obtained from our inference technique with the goal of gaining deeper understanding of MegaD. We present techniques to analyze the protocol models to identify the critical links in botnet C&Cs, design flaws, the existence of background communication channels between C&C servers, and to identify implementation differences for fingerprinting and flaw detection.

Identifying the Critical Links

Transitions in our Mealy machine models represent actions. Certain actions might be considered as *bad*, in the sense that they represent a malicious or undesirable activity. When Mealy machines are used, such activities are represented with transitions (more precisely, output responses).[8] Once the bad transitions are identified, we wish to find a way to prevent such transitions from ever being executed.

More formally, given a protocol state-machine $M = (Q, \Sigma_I, \Sigma_O, \phi, \lambda, q_0)$ and a set of bad output symbols $B \subseteq \Sigma_O$ representing bad actions, we wish to identify the minimal number of transitions we have to disrupt to prevent the bots from executing transitions that would produce output symbols from B. There are two ways (not mutually exclusive) of achieving this. The first option is to make it impossible for bots to reach the states from which bad actions could be performed by cutting a set of transitions in the state-machine, i.e., we wish to assure that $\forall s \in \Sigma_I^*, a \in \Sigma_I \,.\, \lambda\left(\phi^*\left(q_0, s\right), a\right) \notin B$. The second option is to disrupt the bad transitions themselves, i.e., to remove the transitions so as to assure that the following property holds: $\forall s \in \Sigma_I^* \,.\, \left[\lambda^*\left(q_0, s\right)\right] \cap B = \emptyset$.

Such an analysis can be done using max-flow min-cut algorithms [28], such that the initial state is a source, and the state at which the bad transition originates is a sink. We performed this analysis on the MegaD state-machine and arrived at a trivial conclusion for a single pool of bots: since MegaD is a spamming botnet and a single spamming edge is the only bad edge, taking down any one of the botnet servers and the corresponding transitions in the state-machine would prevent a pool of bots from spamming. However, since different pools of bots talk to different sets of servers, it does not stop other pools of MegaD bots from spamming. Unsatisfied with this outcome, we attempt to develop an approach that works across multiple pools of bots.

We extended the encoding of messages into alphabet symbols shown in Table 3.1 by partitioning the set of messages into disjoint sets, one set per server, so as to include the IP addresses of the servers as an additional field. We shall refer to this extended alphabet as IP-extended. We ran our inference technique independently on both master servers we have access to (each pool of bots

[8] The conversion of Mealy to Moore machines introduces additional states, usually one extra state per transition. Hence the two concepts, bad transitions in Mealy machines and bad states in Moore machines are, as a matter of fact, equivalent concepts. Thus, talking about *bad transitions*, as opposed to *bad states*, makes more sense in our setting.

talks to a different master server), and computed a projection (defined below) of one state-machine onto the IP-extended alphabet of the other.

Definition 9 (State-Machine Projection)**.** *The projection of a finite state-machine* $M = (Q, \Sigma_I, \Sigma_O, \phi, \lambda, q_0)$ *onto alphabet* Σ_A *is defined as a non-deterministic finite state-machine* $M' = (Q, \Sigma_I \cap \Sigma_A, \Sigma_O, \phi', \lambda', q_0)$*, such that the following holds for* $\forall\, a \in \Sigma_I, x \in \Sigma_O, q_i, q_j \in Q$*:*

$$
\begin{aligned}
(q_i, \varepsilon, q_j) \in \phi' \quad &\textit{iff} \quad (q_i, a, q_j) \in \phi \wedge a \notin \Sigma_A \\
(q_i, a, q_j) \in \phi' \quad &\textit{iff} \quad (q_i, a, q_j) \in \phi \wedge a \in \Sigma_A \\
(q_i, \varepsilon, \varepsilon) \in \lambda' \quad &\textit{iff} \quad (q_i, a, x) \in \lambda \wedge a \notin \Sigma_A \\
(q_i, a, x) \in \lambda' \quad &\textit{iff} \quad (q_i, a, x) \in \lambda \wedge a \in \Sigma_A
\end{aligned}
$$

Intuitively, all transitions of M on alphabet symbols not in Σ_A are replaced with non-deterministic transitions (ε), and the corresponding outputs are replaced with empty outputs (ε). The resulting state-machine may be non-deterministic.

Computing a projection of a state-machine inferred from communication of our bot emulator with one master server onto the alphabet of the machine inferred from communication with another master server, we identified the key components shared among multiple pools of bots. The results are presented in Section 3.6.

Identifying Design Flaws

We identified a design flaw in the MegaD protocol, thanks to the fact that our inference approach infers complete state-machines. Given a complete state-machine and a specification (i.e., a set of properties expressed in a suitable formal logic), it is possible to automatically determine whether the properties hold using automatic model checkers (e.g., [25]). In our case, the state-machines were simple enough that we could manually check a number of interesting properties. We explain the flaw we found later in Section 3.6.

Identifying Background-Channels

In situations when a client (a bot, in our case) talks to multiple servers, it might be interesting to prove whether there exists any background communication between the servers. Such background communication channels can indicate infiltration traps, which security researchers need to be aware of before attempting to bring a botnet down, or simply reveal interesting information about the protocol.

To detect the background-channels, we devised the following analysis: We restrict our bot emulator to communicate only with a single server at the time, and infer the protocol model M_T (for the template server), M_S (for the SMTP server), and M_M (for the master server). Then, we allow our bot emulator to communicate with all the servers and compute the model M. We compute the projection (Definition 9) of M onto input alphabets used for building individual server communication models (M_T, M_S, and M_M), and compare the obtained projection with the model of communication

with the individual servers. Any differences imply that there exist background communication channels. We prove existence of communication between MegaD's servers in Section 3.6.

Identifying Implementation Differences

Once the complete models of two different implementations of the same protocol are computed, comparison of the models can reveal interesting deviations useful for fingerprinting and flaw detection. While it is possible to perform automatic equivalence checking of large finite-state models (e.g., [55]), our models were simple enough that we can do such an analysis manually. We discuss the differences between Postfix SMTP 2.5.5 and MegaD's implementation in Section 3.6.

3.6 Experimental Evaluation

We implemented our version of $L*$ in $\sim$ 1.7 KLOC of C++ and the bot emulator and experimental infrastructure in $\sim$ 2.3 KLOC of Python and Bash scripts. Our prototype performs up to eight parallel queries (as shown in Figure 3.3) concurrently tunneled through Tor [32]. We conducted the experiments over a period of three weeks starting March 27^{th}, 2010. Figure 3.5 illustrates the inferred MegaD protocol state-machine.

In the rest of this section, we evaluate our protocol inference approach on the MegaD botnet C&C distributed system, MegaD's non-standard implementation of SMTP, and the standard SMTP as implemented in Postfix 2.5.5. We present the results of our analysis of inferred complete models, and validate our inference approach by comparing the inferred SMTP models against the SMTP standard.

Performance and Accuracy

In this section, we present the experimental evidence of the effectiveness of our response prediction technique and discuss the model accuracy.

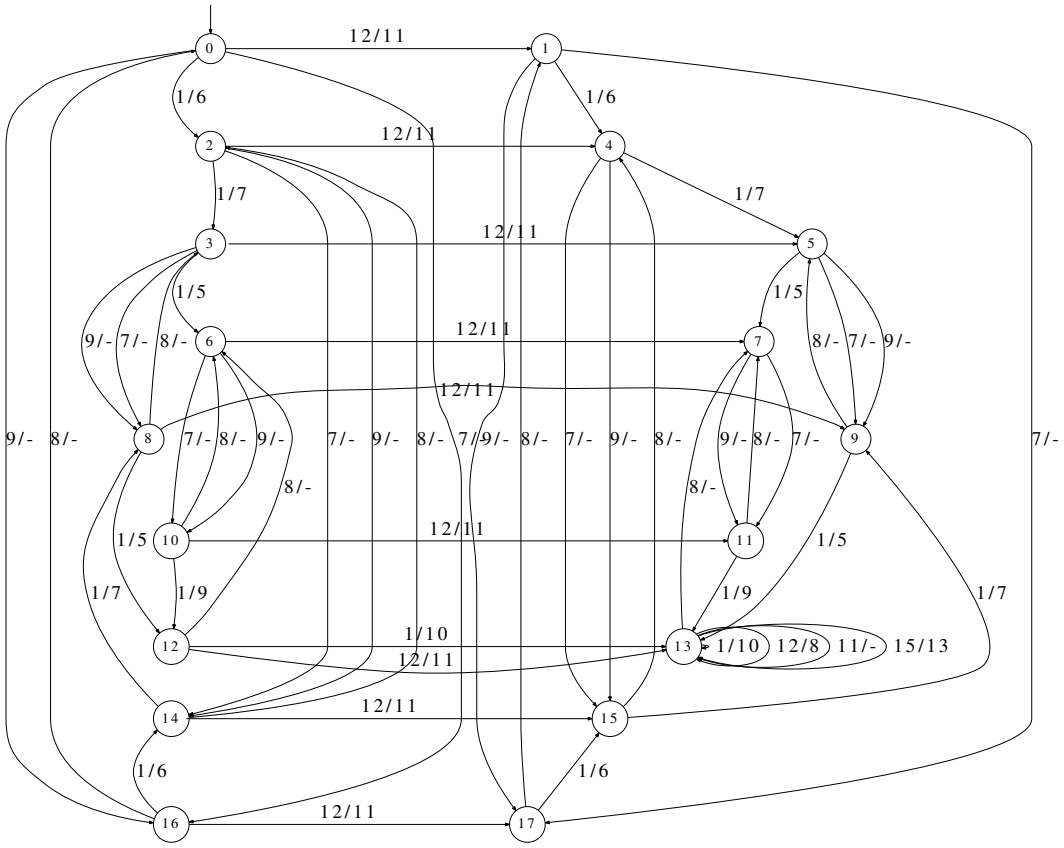


Figure 3.5: Protocol State-Machine of MegaD's distributed Command and Control System. Self-transition edges are removed for clarity. The state-machine transitions are labeled according to the alphabet in Table 3.1. For example, 15/13 denotes NOTIFY / NOTIFY_RECVED. The process of spamming is triggered in state 13, through self-edges 1/10, 12/8, 11/-, 15/13.

The prediction results are shown in Table 3.2. The overall reduction in the number of queries that have to be sent over the network is between 24.5% (for MegaD's SMTP) and 86.1% (for MegaD's C&C). We believe there are two main reasons our prediction is much more effective on the MegaD C&C than on SMTP: First, C&C is a more complex protocol that involves three different types of servers, two of which use proprietary protocols. Second, our understanding of the two protocols when we were designing message abstractions was very different — we knew nothing about the C&C state-machine, while the SMTP state-machine is well known [53]. We believe this inherent lack of knowledge about an unknown protocol model, yet to be inferred, results in some amount of redundancy. However, it is important to be conservative when abstracting messages, as otherwise it is easy to miss important states and transitions. This inherent tradeoff between accuracy and redundancy makes our prediction technique even more valuable, as we can infer larger protocols, without sacrificing accuracy. As a matter of fact, since sending and receiving

	MegaD C&C		MegaD SMTP		STD SMTP	
	Q.	Msgs	Q.	Msgs	Q.	Msgs
Basic $L*$	10,978	56,716	1,190	4,522	1,386	5,894
RESTR	-8,024	-42,872	-294	-980	-476	-1764
STAT	-1,456	-7,514	-22	-88	-0	-0
BCKT	+24	+76	+24	+90	+56	+252
Total	1,522	6,406	898	3,544	966	4,382
Reduct.	-86.1%	-88.7%	-24.5%	-21.6%	-30.3%	-25.7%
Accur.	99.7%	99.9%	92.4%	97.8%	88.2%	96.8%

Table 3.2: Results of Membership Queries Prediction. The Queries (Q.) column shows the number of queries and the Msgs column shows the number of messages. The first row represents the results obtained in the standard $L*$ algorithm [82] without any response prediction. The RESTR row shows the reduction in the number of queries and messages by using restriction-based prediction. The following row (STAT) shows additional reductions obtained by using the statistics-based approach a top of RESTR. The BCKT row shows the increase in the number of queries and messages sent due to backtracking, caused by prediction errors. The Total row shows the total numbers of queries and messages after prediction reductions and increases due to backtracking. The total reduction in the number of queries and messages and the accuracy of the prediction are shown in the last two rows.

a single message through Tor took around 6.8 seconds on average, 86.1% prediction accuracy means that our response predictor saved $\frac{(56{,}716-6{,}406)\times 6.8}{3600\times 24} = 3.95$ days of computation, reducing the total amount of time required to infer the MegaD C&C to around 12 hours.

Parallelization of the experiment improved the performance even further. While a single bot emulator would return a response message every 6.8 seconds, on average, eight parallel bots would return a message every 1.4 seconds, a 4.85X improvement on average in addition to improvements obtained by response prediction. The experiment does not parallelize perfectly, because of the increased load on the same Tor servers. We envision that a more powerful, perhaps even highly-distributed, protocol inference infrastructure would parallelize even better.

To check the accuracy of our models, we used $\varepsilon = 10^{-2}$ error probability and $\gamma = 10^{-6}$ confidence factors [4]. Achieving that accuracy required us to generate at least 1,798 uniformly random sampling queries for the MegaD C&C and 1,451 for MegaD and Postfix's SMTP upon $L*$ termination. Equivalence queries were ran in parallel and cached, but not predicted, as predicting those queries would defeat the purpose of such queries (detecting missed states and prediction errors).

Analysis of Critical Links

Attempts to bring down large botnets are frequent, but are costly and ineffective. The common practice is to run as many pools of bots as possible to obtain a wide coverage of C&C servers, and then attempt to cripple the entire botnet by sending abuse notifications to all ISPs involved

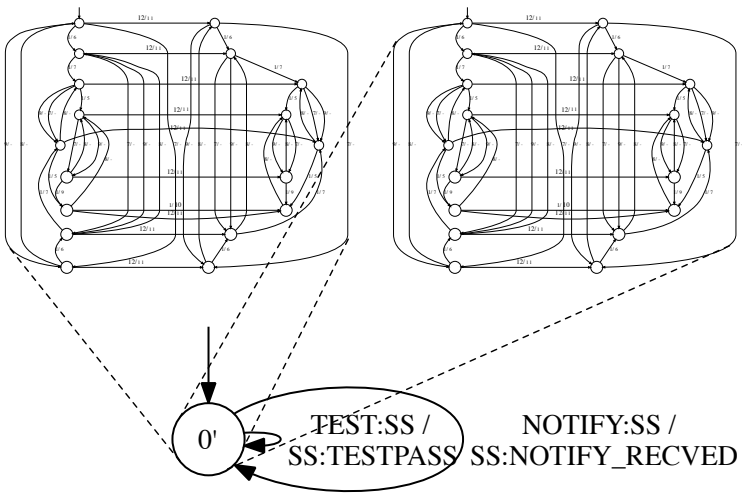


Figure 3.6: Intersection of Models of Communication with Two Different Master Servers. The resulting state-machine shows that the SMTP server is shared among two pools of bots communicating with two different master servers.

simultaneously. This is an expensive exercise requiring careful co-ordination among all parties involved. Based on our analysis of critical links in the MegaD protocol, we discovered a less expensive alternative.

Using the technique explained in Section 3.5, we inferred complete models of communication with MegaD's two different master servers, and computed a projection of one model onto the alphabet of the other model as shown in Figure 3.6. The figure shows the states and links shared by two different pools of bots talking to *different* master servers and the servers that the master server points to. The projection shows that the SMTP server is shared across two pools of bots belonging to different master servers. Furthermore, the existence of the SMTP server is critical to MegaD's ability to spam. In particular, a successful pre-spam notification dialog with the shared SMTP server (the NOTIFY:SS/SS:NOTIFY_RECVED edge) *always* signals to the bot to start spamming. We experimentally confirmed that MegaD bots do not start spamming without this notification. Thus, the analysis clearly shows that taking down the SMTP server would disable spamming in all the pools of bots sharing that SMTP server. However, without actually attempting to execute an attack on the SMTP part of MegaD's infrastructure, it is difficult to evaluate how useful our insight is in practice. For example, the botmasters could replace the SMTP servers and use master servers to update bots on the new SMTP server locations.

Analysis of Design Flaws

Having a complete model of MegaD's C&C enabled us to check a number of properties on the inferred state-machine. In particular, we found that there is an unexpectedly short path through the

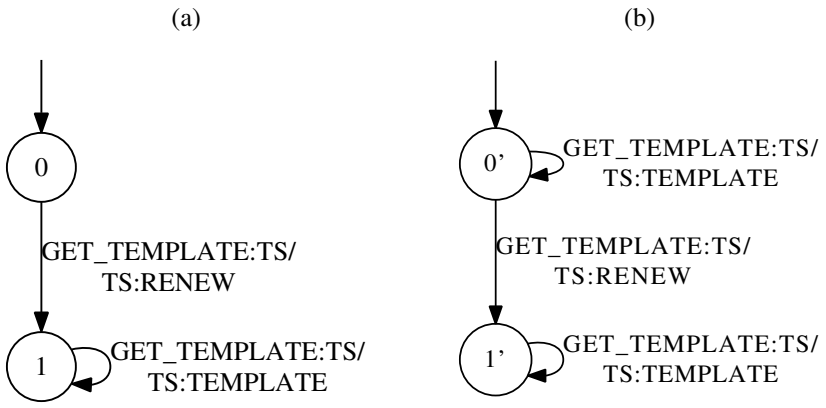


Figure 3.7: A Proof of the Existence of Background-Channel Communication. (a) A Model Obtained Only from the Communication with the Template Server. (b) The Projection of the C&C State-Machine onto the Template Server Alphabet. States {0,2,3,6,7,8,10,11,12,13,14,16} and {1,4,5,9,15,17} of the C&C in Figure 3.5 are projected onto states 0' and 1' respectively. Notice the state-machine is non-deterministic.

state-machine to getting the templates from the template sever.

During a normal spam cycle, a MegaD bot would take the following path $0 \rightarrow 16 \rightarrow 14 \rightarrow 8 \rightarrow 12 \rightarrow 13$ to the spamming state 13. Upon reaching state 13, the bot sends the GET_COMMAND message, to which the master server responds with the location of a chosen template server. The bot then sends the GET_TEMPLATE request to the template server together with a 16-byte bot identifier issued by the master server, and receives templates as a response.

Our inferred C&C protocol model reveals the existence of multiple shorter paths to obtaining spam templates. In particular, the shortest path is $0 \rightarrow 1$, along which the bot emulator bypasses the master server, directly contacts the template server with a random 16-byte identifier, and obtains the templates while bypassing the master server. We successfully exploited this protocol design flaw, regularly obtaining fresh spam templates.

Without knowing the botnet developer's intentions, one may argue that what we discover may be a "feature" instead of a "flaw". For instance, the "feature" might facilitate reconfiguration of the template server location. However, we rule out such a possibility because the protocol uses an encryption-protected 16-byte identifier issued by the master server in the GET_TEMPLATE request. Since this identifier can be spoofed, it is clearly a flaw.

Analysis of Background-Channels

Our analysis discovered that the template server behaves differently depending on whether the bot communicated with the master server or not. If the bot talks only to the template server (Figure 3.7a) by sending a GET_TEMPLATE request, the template server responds with RENEW, and sends the templates only after the second request. On the other hand, if the bot talks to the template server after the regular message exchange with the master server, the template responds to the first

GET_TEMPLATE request immediately with spam templates. This difference proves that there exists communication between the master and template servers. The model of this communication is shown in Figure 3.7b.

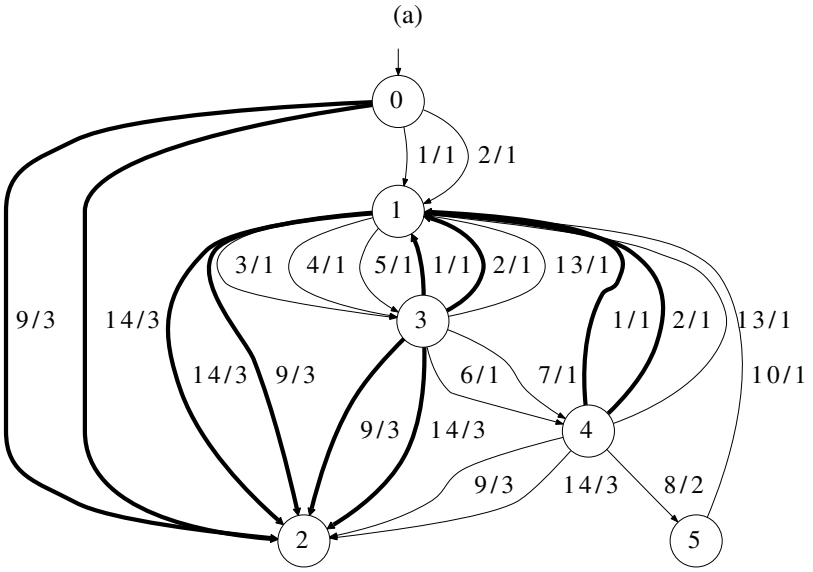
Analysis of Implementation Differences

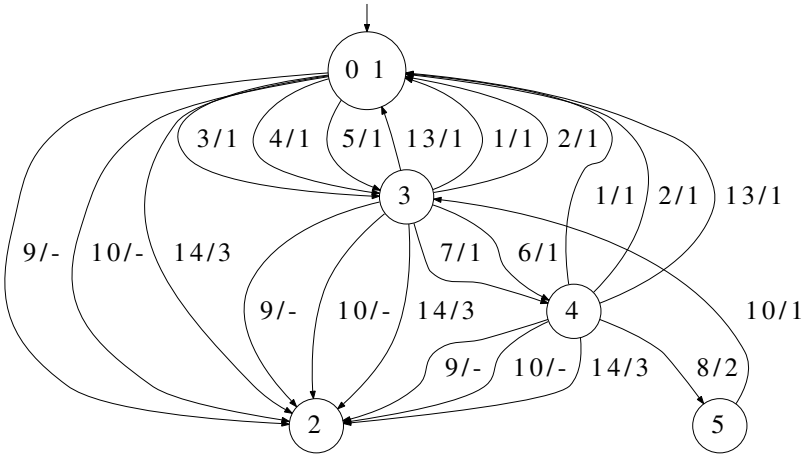
We cross-checked the results of our inference with the SMTP standard [53], and compared the inferred model with the MegaD's non-standard SMTP implementation. To test the standard SMTP inference, we set up a Postfix SMTP 2.5.5 server on Mac OS X 10.6.3 and ran our inference technique against the server, using the standard set of 14 SMTP commands and input types from the SMTP specification [53], shown in Figure 3.8c. Figure 3.8a shows the inferred protocol model. For clarity, we removed all self-loop transition edges in the figure. As discussed in Section 3.6, the discrepancy is at most $\varepsilon = 10^{-2}$ with confidence $\gamma = 10^{-6}$.

Our evaluation on standard SMTP shows that the protocol model obtained from our inference technique is equivalent to the SMTP standard, with one implementation-specific deviation. We found and confirmed that the Postfix SMTP 2.5.5 implementation deviates from the SMTP standard by terminating a connection with response code '221' when it receives email contents sent without a prior DATA command; the standard specifies keeping the connection open while returning '500' to indicate an unrecognized command. We further evaluated our model by comparing it against the SMTP model inferred by Prospex [27]. We manually translated our inferred Mealy machine into Prospex's representation (Moore machine). The comparison reveals that a number of edges are missing from the SMTP model that Prospex inferred. On inter-state transitions alone, our complete SMTP model specifies the behavior of ten edges that Prospex missed. For example, the behavior of sending email contents without prior HELO or EHLO is described in our model, but not in [27]. We bold these missing edges in Figure 3.8a. Clearly, our technology is able to infer complete models, unlike the prior work.

Our model completely captures the protocol and is smaller (5 states) than the incomplete Prospex model (10 states). There are two reasons for this. First, we use Mealy machines, which are more compact. Second, since we infer complete minimal machines, minimization of the state-machine can be done completely and thoroughly. For example, sending either HELO or EHLO from the initial state transits to a single state in our model instead of two different states.

We also compared MegaD's non-standard SMTP using the same standard set of 14 SMTP commands and input types that we used to infer the standard SMTP protocol model. The inferred MegaD non-standard SMTP protocol model is shown in Figure 3.8b. A comparison of Figures 3.8a and 3.8b yields a high degree of resemblance, with two major differences: (1) States 0 and 1 in Figure 3.8a are merged into a single state in Figure 3.8b, which indicates that the usual SMTP dialog with MegaD's SMTP server may take place without a prior HELO or EHLO. (2) The server abruptly closes the connection once it receives a data terminator '.' without a prior DATA command. This result suggests that our protocol model inference technique may be applied to fingerprint and identify MegaD's SMTP masquerading servers.





(c)

ID	Semantics	Direction	ID	Semantics	Direction
1	HELO 1	$\rightarrow$ SS	10	\r\n.\r\n	$\rightarrow$ SS
2	EHLO 1	$\rightarrow$ SS	11	EXPN all	$\rightarrow$ SS
3	MAIL FROM:$< addr >$	$\rightarrow$ SS	12	VRFY usr	$\rightarrow$ SS
4	MAIL FROM:<>	$\rightarrow$ SS	13	RSET	$\rightarrow$ SS
5	MAIL FROM:$< @a, @b : addr >$	$\rightarrow$ SS	14	QUIT	$\rightarrow$ SS
6	RCPT TO:$< addr >$	$\rightarrow$ SS	1	250	$\leftarrow$ SS
7	RCPT TO:$< @a, @b : addr >$	$\rightarrow$ SS	2	354	$\leftarrow$ SS
8	DATA	$\rightarrow$ SS	3	221	$\leftarrow$ SS
9	content	$\rightarrow$ SS			

Figure 3.8: Inferred SMTP State Machines: (a) Postfix SMTP 2.5.5, (b) MegaD non-Standard SMTP and (c) Abstraction Table.

3.7 Limitations

In this section, we highlight the limitations of our protocol model inference and analysis techniques, and discuss possible directions for future work.

Currently, we make no effort to obfuscate and hide our probing traffic from the botmaster's possible detection. Since our protocol inference approach generates a large number of probing queries to the botnet C&C servers, the botmaster could potentially detect our queries. The botmaster may react by changing the locations or protocol configurations of the C&C servers, thwarting our inference effort. Hiding our traffic in the background noise would be an interesting area for future research.

We also note that our protocol state-machine inference approach would not work if any assumptions laid out in Section 3.2 are violated. The determinism and finiteness assumptions are the most limiting.

A protocol that behaves non-deterministically (e.g., a date/time triggered behavior) is more challenging to infer. As discussed in Section 3.2, one option is to discretize time and encode it directly into the alphabet. Despite obvious limitations, our conjecture is that such an approach could be sufficient for inferring a majority of existing protocols. Only future research can (dis)prove our conjecture.

While infinite-state protocols can be abstracted with finite-state machines (c.f. Mohri-Nederhof algorithm for abstracting context-free with regular ones [68]), such abstractions might not be precise enough for all potential applications. To make things worse, many protocols have mildly-context-sensitive features, like buffer lengths. The grammatical inference techniques [45] for such more expressive languages are still in their infancy. Inference of more expressive models is a promising research direction, not only in the context of protocol inference.

The focus of our work is the model inference technique itself, and we relied upon prior work [14] and manual abstraction to come up with the alphabet, which might be incomplete, i.e., it might not contain all messages that can cause a state-change in the protocol. The clustering of messages in a single direction for abstraction can be automatic [27], but cannot guarantee the completeness of the alphabet either. To our knowledge, automatic reverse-engineering of the complete alphabet is an open problem.

3.8 Related Work

The work presented in this thesis is in the intersection of protocol model inference and grammatical inference, and also contributes to the previous work in the area of analysis of botnets (e.g., [2, 38, 39, 40, 50]).

The most closely related work, to our knowledge, is the work of Comparetti et al. [27] on the Prospex system. The automatic clustering and abstraction feature in Prospex is more advanced than the manual abstraction that we did, so our future work is likely going to focus on improving that aspect of our system. Unlike our approach, Prospex adopts passive off-line inference and models protocols with Moore machines. These choices have a number of consequences: (1) In-

ferred models are incomplete state-machines, which means that any subsequent analysis is bound to be imprecise. In contrast, our approach infers complete state machines. (2) Since protocols are reactive systems, there are no accepting or rejecting states. To work around that problem, Prospex differentiates states using regular expressions describing messages received before each state is reached. The regular expression labels prevent the incomplete state-machine minimization algorithms, like Exbar [57], to merge all the states into a single state. We avoid this problem altogether by using a model more appropriate for reactive systems — Mealy machines and $L*$ algorithm guarantee that the inferred machine is minimal. (3) The minimization of incomplete state-machines is a known NP-complete problem [76], so it is questionable whether Prospex would scale to large complex protocols. Our approach dodges the NP-completeness using proactive inference, at the cost of a small probability of error ($\varepsilon = 10^{-2}$) with high confidence ($\gamma = 10^{-6}$) [4].

Earlier work on protocol inference by Hsu et al. [48] does use Mealy machines, but also adopts off-line inference, which means that the inferred models will be incomplete and minimization is NP-complete. Their solution is an approximate algorithm. Unfortunately, even computing a model that is within a polynomial size of the minimal one is NP-complete [88, p. 98–99], meaning that their polynomial-time approximation algorithm will compute very large models in some cases. In contrast, we developed a version of $L*$ optimized for on-line protocol inference, with known advantages over the off-line approaches (polynomial computational complexity, completeness).

The results presented in this chapter would not have been possible without the prior research on automatic message format reverse-engineering by Cui et al. [29, 30] and the work of Caballero et al. [15], which we used in this chapter. All these techniques are crucial for both manual and automatic message abstraction into finite alphabets, so we are looking forward to the further progress in research on automatic message format reverse-engineering.

Another aspect important in general protocol model inference is dealing with encryption. Caballero et al. [14] recently proposed an automatic technique for extracting encryption routines from binary. Similarly, Wang et al. [92] deals with reverse-engineering of encrypted messages.

Once the protocol model is known, it can be used to incorporate into stateful protocol analyzers, like Bro [74] and GAPA [10], and firewalls, like Shield [91]. All these systems require protocol specifications to analyze traffic, detect intrusions, and improve security. The technology we developed can provide such specifications.

Our contributions to the field of grammatical inference (more precisely, regular language inference) extend the previous work of Shahbaz and Groz [82], by specializing their approach to protocol inference, and by a number of optimizations for decreasing the number of membership queries, which are expensive in the real network setting. While parallelization of $L*$ and introduction of a cache for concentrating results of parallel probes came as natural optimizations suitable for our setting, the output symbol prediction required more intellectual effort. Our inspiration came from the recent work of Gupta and McMillan [43]. They applied decision-trees [78], a standard machine-learning technique, to complete incomplete state-machines learned in the hardware verification setting for the purpose of abstracting hardware modules and achieving compositional verification.

3.9 Conclusions

We have proposed, to the best of our knowledge, the first technique to infer complete protocol state machines in the realistic high-latency network setting, and applied it to the analysis of botnet C&C protocols. While the classic $L*$ algorithm would have taken 4.46 days to infer the protocol model of the MegaD C&C distributed system, we introduced a novel and effective prediction technique to minimize the number of queries generated during the inference process, reducing the amount of time required to just 12 hours. This is further optimized through parallelization.

By analyzing the complete protocol models inferred by our technique, we offer novel insights to existing problems on botnet C&Cs. We hope that our new insights, gained through our protocol inference technique and novel analyses, will render future attacks on MegaD and other botnets cheaper and more effective. With the new technology to fight botnets we developed, we hope to see a decrease in the amount of spam and denial of service attacks, and an increase of everyone's productivity and security.

Chapter 4

Synergies between Model Inference and Symbolic Execution

4.1 Introduction

Designing secure systems is an exceptionally hard problem. Even a single bug in an inopportune place can create catastrophic security gaps. Considering the size of modern software systems, often reaching tens of millions of lines of code, exterminating all the bugs is a daunting task. Thus, innovation and development of new tools and techniques that help closing security gaps is of critical importance. In this chapter, we propose a new technique for exploring the program's state-space. The technique explores the program execution space automatically by combining exploration with learning of an abstract model of program's state space. More precisely, it alternates (1) a combination of concrete and symbolic execution [52] to explore the program's state-space, and (2) the L^* [4] online learning algorithm to construct high-level models of the state-space. Such abstract models, in turn, guide further search. In contrast, the prior state-space exploration techniques treat the program as a flat search-space, without distinguishing states that correspond to important input processing events.

A combination of concrete execution and symbolic reasoning, known as DART, concolic (*conc*rete and symb*olic*) execution, and dynamic symbolic execution [34, 80, 17, 16], exploits the strengths of both. The concrete execution creates a path, followed by symbolic execution, which computes a symbolic logical formula representing the branch conditions along the path. Manipulation of the formula, e.g., negation of a particular branch predicate, produces a new symbolic formula, which is then solved with a decision procedure. If a solution exists, the solution represents an input to the concrete execution, which takes the search along a different path. The process is repeated iteratively until the user reaches the desired goal (e.g., number of bugs found, code coverage, etc.).

We identified two ways to improve this iterative process. First, dynamic symbolic execution has no high-level information about the structure of the overall program state-space. Thus, it has no way of knowing how close (or how far) it is from reaching important states in the program and is

likely to get stuck in local state-subspaces, such as loops. Second, unlike decision procedures that learn search-space pruning lemmas from each iteration (e.g., [93]), dynamic symbolic execution only tracks the most promising path prefix for the next iteration [34], but does not learn in the sense that information gathered in one iteration is used either to prune the search-space or to get to interesting states faster in later iterations.

These two insights led us to develop an approach — Model-inference-Assisted Concolic (*conc*rete and symb*olic*) Exploration (MACE) — that learns from each iteration and constructs a finite-state model of the search-space. We primarily target applications that maintain an ongoing interaction with its environment, like servers and web services, for which a finite-state model is frequently a suitable abstraction of the communication protocol, as implemented by the application. At the same time, we both learn the protocol model and exploit the model to guide the search.

MACE relies upon dynamic symbolic execution to discover the protocol messages, uses a special filtering component to select messages over which the model is learned, and guides further search with the learned model, refining it as it discovers new messages. Those three components alternate until the process converges, automatically inferring the protocol state machine and exploring the program's state-space.

We have implemented our approach and applied it to four server applications (two SMB and two RFB implementations). MACE significantly improved the line coverage of the analyzed applications, and more importantly, discovered four new vulnerabilities and three known ones. One of the discovered vulnerabilities received Gnome's "Blocker" severity, the highest severity in their ranking system meaning that the next release cannot be shipped without a fix. Our work makes the following contributions:

- Although dynamic symbolic execution and decision procedures perform very similar tasks, the state-of-the-art decision procedures feature many techniques, like learning, that yet have to find their way into dynamic symbolic execution. While in decision procedures, learned information can be conveniently represented in the same format as the solved formula, e.g., in the form of CNF clauses in SAT solvers, it is less clear how would one learn or represent the knowledge accumulated during the dynamic symbolic execution search process. We propose that for applications that interact with their environment through a protocol, one could use finite-state machines to represent learned information and use them to guide the search.
- As the search progresses, it discovers new information that can be used to refine the model. We show one possible way to keep refining the model by closing the loop — search incrementally refines the model, while the model guides further search.
- At the same time, MACE both infers a model of the protocol, as implemented by a program, and explores the program's search space, automatically generating tests. Thus, our work contributes both to the area of automated reverse-engineering of protocols and automated program testing.
- MACE discovered seven vulnerabilities (four of which are new) in four applications that we analyzed. Furthermore, we show that MACE performs deeper state-space exploration than the baseline dynamic symbolic execution approach.

4.2 Related Work

Model-guided testing has a long history. The hardware testing community has developed modeling languages, like SystemVerilog, that allow verification teams to specify input constraints that are solved with a decision procedure to generate random inputs. Such inputs are randomized, but adhere to the specified constraints and therefore tend to reach much deeper into the tested system than purely random tests. Constraint-guided random test generation is nowadays the staple of hardware testing. The software community developed its own languages, like Spec# [7], for describing abstract software models. Such models can be used effectively as constraints for generating tests [89], but have to be written manually, which is both time consuming and requires a high level of expertise.

Grammar inference (e.g., [45]) promises automatic inference of models, and has been an active area of research in security, especially applied to protocol inference. Comparetti et al. [27] infer incomplete (possibly missing transitions) protocol state machines from messages collected by observing network traffic. To reduce the number of messages, they cluster messages according to how similar the messages are and how similar their effects are on the execution. Comparetti et al. show how the inferred protocol models can be used for fuzzing. Our work shares similar goals, but features a few important differences. First, MACE iteratively refines the model using dynamic symbolic execution [35, 80, 18, 16] for the state-space exploration. Second, rather than filtering out individual messages through clustering of individual messages, we look at the entire sequences. If there is a path in the current state machine that produces the same output sequence, we discard the corresponding input sequence. Otherwise, we add all the input messages to the set used for inferring the state machine in the next iteration. Third, rather than using the inferred model for fuzzing, we use the inferred model to initialize state-space exploration to a desired state, and then run dynamic symbolic execution from the initialized state.

In our prior work [20], we proposed an alternative protocol state machine inference approach. There we assume the end users would provide abstraction functions that abstract concrete input and output messages into an abstract alphabet, over which we infer the protocol. Designing such abstraction functions is sometimes non-trivial and requires multiple iterations, especially for proprietary protocols, for which specifications are not available. In this work, we drop the requirement for user-provided input message abstraction, but we do require a user-provided output message abstraction function. The output abstraction function determines the granularity of the inferred abstraction. The right granularity of abstraction is important for guiding state-space exploration, because too fine-grained abstractions tend to be too expensive to infer automatically, and too abstract ones fail to differentiate interesting protocol states. Furthermore, our prior work in the previous chapter is a purely black-box approach, while in this chapter we do code analysis at the binary level in combination with grammatical inference.

In this chapter, we analyze implementations of protocols for which the source code or specifications are available. However, MACE could also be used for inference of proprietary protocols and for state-exploration of closed-source third-party binaries. In that case, the users would need to rely upon the prior research to construct a suitable output abstraction function. The first step in constructing a suitable output abstraction function is understanding the message format. Cui

et al. [29, 30] and Caballero et al. [15] proposed approaches that could be used for that purpose. Further, any automatic protocol inference technique has to deal with encryption. In this chapter, we simply configure the analyzed server applications so as to disable encryption, but that might not be an option when inferring a proprietary protocol. The work of Caballero et al. [14] and Wang et al. [92] addresses automatic reverse-engineering of encrypted messages.

Software model checking tools, like SLAM [6] and Blast [44], incrementally build predicate abstractions of the analyzed software, but such abstractions are very different from the models inferred by the protocol inference techniques [27, 21]. Such abstractions closely reflect the control-flow structure of the software from which they were inferred, while our inferred models are more abstract and tend to have little correlation with the low-level program structure. Further, depending on the inference approach used, the inferred models can be minimal (like in our work), which makes guidance of state-space exploration techniques more effective.

The Synergy algorithm [41] combines model-checking and dynamic symbolic execution to try to cover all abstract states of a program. Our work has no ambition to produce proofs, and we expect that our approach could be used to improve the dynamic symbolic execution part of Synergy and other algorithms that use dynamic symbolic execution as a component.

The Ketchum approach [46] combines random simulation to drive a hardware circuit into an interesting state (according to some heuristic), and performs local bounded model checking around that state. After reaching a predefined bound, Ketchum continues random simulation until it stumbles upon another interesting state, where it repeats bounded model checking. Ketchum became the key technology behind Magellan™, one of the most successful semi-formal hardware test generation tools. MACE has similar dynamics, but the components are very different. We use the L^* [4] finite-state machine inference algorithm to infer a high-level abstract model and declare all the states in the model as interesting, while Ketchum picks interesting states heuristically. While Ketchum uses random simulation, we drive the analyzed software to the interesting state by finding the shortest path in the abstract model. Ketchum explores the vicinity of interesting states via bounded model checking, while we start dynamic symbolic execution from the interesting state.

4.3 Problem Definition and Overview

We begin this section with the problem statement and a list of assumptions that we make in this chapter. Next, we discuss possible applications of MACE. At the end of this section, we introduce the concepts and notation that will be used throughout the chapter.

Problem Statement

We have three, mutually supporting, goals. First, we wish to automatically infer an abstract finite-state model of a program's interaction with its environment, i.e., a protocol as implemented by the program. Second, once we infer the model, we wish to use it to guide a combination of concrete and symbolic execution in order to improve the state-space exploration. Third, if the exploration phase discovers new types of messages, we wish to refine the abstract model, and repeat the process.

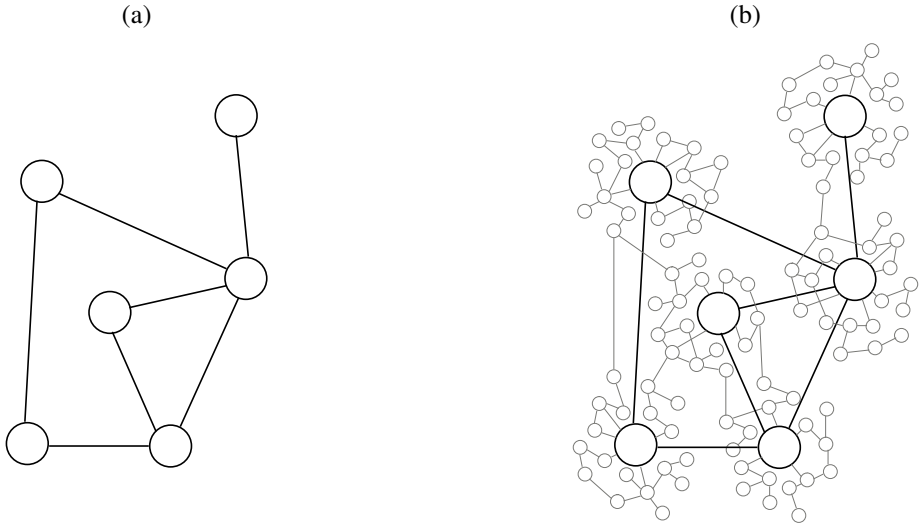


Figure 4.1: An Abstract Rendition of the MACE State-Space Exploration. The figure on the left shows an abstract model, i.e., a finite-state machine, inferred by MACE. The figure on the right depicts clusters of concrete states of the analyzed application, such that clusters are abstracted with a single abstract state. We infer the abstract model with L^*, initialize the analyzed application to the desired state, and then use the state-space exploration component of MACE to explore the concrete clusters of states.

There are two ways to refine the abstract finite-state model; by adding more states, and by adding more messages to the state machine's input (or output) alphabet, which can result in inference of new transitions and states. Black box inference algorithms, like L^* [4], infer a state machine over a fixed-size alphabet by iteratively discovering new states. Such algorithms can be used for the first type of refinement. Any traditional program state-space exploration technique could be used to discover new input (or output) messages, but adding all the messages to the state machine's alphabets would render the inference computationally infeasible. Thus, we also wish to find an effective way to reduce the size of the alphabet, without missing states during the inference.

The constructed abstract model can guide the search in many ways. The approach we take in this chapter is to use the abstract model to generate a sequence of inputs that will drive the abstract model and the program to the desired state. After the program reaches the desired state, we explore the surrounding state-space using a combination of symbolic and concrete execution. Through such exploration, we might visit numerous states that are all abstracted with a single state in the abstract model and discover new inputs that can refine the abstract model. Figure 4.1 illustrates the concept.

In our work, we make a few assumptions:

Determinism We assume the analyzed program's communication with its environment is deterministic, i.e., the same sequence of inputs always leads to the same sequence of outputs and

the same state. In practice, programs can exhibit some non-determinism, which we are abstracting away. For example, the same input message could produce two different outputs from the same state. In such a case, we put both output messages in the same equivalence class by adjusting our output abstraction (see below).

Resettability We assume the analyzed program can be easily reset to its initial state. The reset may be achieved by restarting the program, re-initializing its environment or variables, or simply initiating a new client connection. In practice, resetting a program is usually straightforward, since we have a complete control of the program.

Output Abstraction Function We assume the existence of an output abstraction function that abstracts concrete response (output) messages from the server into an abstract set of messages (alphabet) used for state machine inference. In practice, this assumption often reduces to manually identifying which sub-fields of output messages will be used to distinguish output message types. The output alphabet, in MACE, determines the granularity of abstraction.

Applications

The primary intended application of MACE is state-space exploration of programs communicating with their environment through a protocol, e.g., networked applications. We use the inferred protocol state machine as a map that tells us how to quickly get to a particular part of the search-space. In comparison, model checking and dynamic symbolic execution approaches consider the application's state-space flat, and do not attempt to exploit the structure in the state machine of the communication protocol through which the application communicates with the world. Other applications of MACE include proprietary protocol inference, extension of the existing protocol test suites, conformance checking of different protocol implementations, and fingerprinting of implementation differences.

Preliminaries

Following our prior work [20], we use *Mealy machines* [67] as abstract protocol models. Mealy machines are natural models of protocols because they specify transition and output functions in terms of inputs. Mealy machines are defined as follows:

$\hat{=}$[Mealy Machine] A Mealy machine, M, is a six-tuple $(Q, \Sigma_I, \Sigma_O, \phi, \lambda, q_0)$, where Q is a finite non-empty set of states, $q_0 \in Q$ is the initial state, Σ_I is a finite set of input symbols (i.e., the input alphabet), Σ_O is a finite set of output symbols (i.e., the output alphabet), $\phi : Q \times \Sigma_I \longrightarrow Q$ is the transition relation, and $\lambda : Q \times \Sigma_I \longrightarrow \Sigma_O$ is the output relation.

We extend the ϕ and λ relations to sequences of messages $m_j \in \Sigma_I$ as usual, e.g., $\phi(q, m_0 \cdot m_1 \cdot m_2) = \phi(\phi(\phi(q, m_0), m_1), m_2)$ and $\lambda(q, m_0 \cdot m_1 \cdot m_2) = \lambda(q, m_0) \cdot \lambda(\phi(q, m_0), m_1) \cdot \lambda(\phi(q, m_0 \cdot m_1), m_2)$. To denote sequences of input (resp. output) messages we will use lower-case letters s, t (resp. o). For $s \in \Sigma_I^*, m \in \Sigma_I$, the *length* $|s|$ is defined inductively: $|\varepsilon| = 0, |s \cdot m| = |s| + 1$, where ε is the empty sequence. The j-th message m_j in the sequence $s = m_0 \cdot m_1 \cdots m_{n-1}$ will be referred to as s_j. We define the support function *sup* as $sup(s) = \{s_j \mid 0 \leq j < |s|\}$. If for some

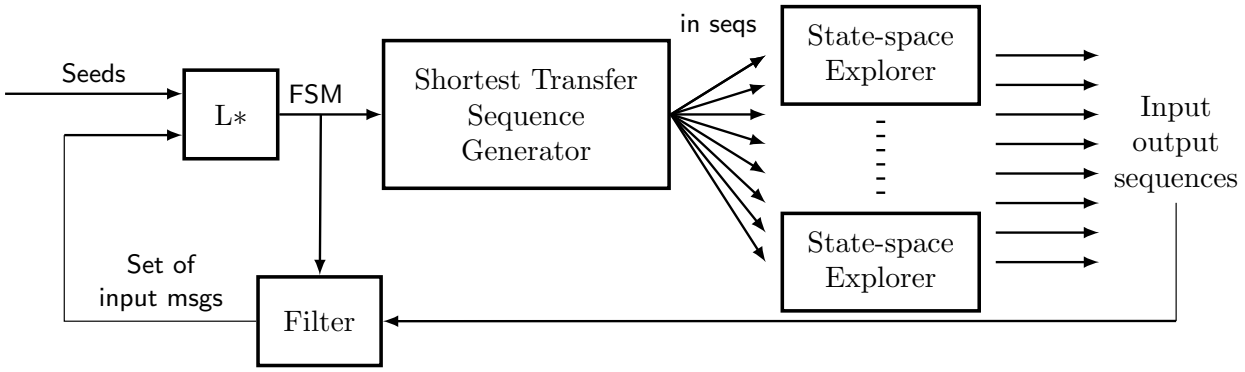


Figure 4.2: The MACE Approach Diagram. The L^* algorithm takes in the input and output alphabets, over which it infers a state-machine. L^* sends queries and receives responses from the analyzed application, which is not shown in the figure. The result of inference is a finite-state machine (FSM). For every state in the inferred state machine, We generate a shortest transfer sequence (Section 4.3) that reaches the desired state, starting from the initial state. Such sequences are used to initialize the state-space explorer, which runs dynamic symbolic execution after the initialization. The state-space explorers run the analyzed application (not shown) in parallel.

state machine $M = (Q, \Sigma_I, \Sigma_O, \phi, \lambda, q_0)$ and some state $q \in Q$ there is $s \in \Sigma_I^*$ such that $\phi(q_0, s) = q$, we say there is a path from q_0 to q, i.e., that q is reachable from the initial state, denoted $q_0 \xrightarrow{*} q$. Since L^* infers minimal state machines, all states in the abstract model are reachable. In general, each state could be reachable by multiple paths. For each state q, we (arbitrary) pick one of the shortest paths formed by a sequence of input messages s, such that $q_0 \xrightarrow{s} q$, and call it a *shortest transfer sequence*.

Our search process discovers numerous input and output messages, and using all of them for the model inference would not scale. Thus, we heuristically discard redundant input messages, defined as follows:

$\hat{=}$[Redundant Input Symbols] Let $M = (Q, \Sigma_I, \Sigma_O, \phi, \lambda, q_0)$ be a Mealy machine. A symbol $m \in \Sigma_I$ is said to be redundant if there exists another symbol, $m' \in \Sigma_I$, such that $m \neq m'$ and $\forall q \in Q \,.\, \lambda(q, m) = \lambda(q, m') \wedge \phi(q, m) = \phi(q, m')$.

We say that a Mealy machine $M = (Q, \Sigma_I, \Sigma_O, \phi, \lambda, q_0)$ is complete iff $\phi(q, i)$ and $\lambda(q, i)$ are defined for every $q \in Q$ and $i \in \Sigma_I$. In this thesis, we infer complete Mealy machines. There is also another type of completeness — the completeness of the input and output alphabet. MACE cannot guarantee that the input alphabet is complete, meaning that it might not discover some types of messages required to infer the full state machine of the protocol.

To infer Mealy machines, we use Shahbaz and Groz's [82] variant of the classical L^* [4] inference algorithm. We describe only the intuition behind L^*, as the algorithm is well-described in the literature.

L^* is an online learning algorithm that proactively probes a black box with sequences of messages, listens to responses, and builds a finite state machine from the responses. The black box is

expected to answer the queries in a faithful (i.e., it is not supposed to cheat) and deterministic way. Each generated sequence starts from the initial state, meaning that L^* has to reset the black box before sending each sequence. Once it converges, L^* conjectures a state machine, but it has no way to verify that it is equivalent to what the black box implements. Three approaches to solving this problem have been described in the literature. The first approach is to assume an existence of an *oracle* capable of answering the *equivalence queries*. L^* asks the oracle whether the conjectured state machine is equivalent to the one implemented by the black box, and the oracle responds either with 'yes' if the conjecture is equivalent, or with a counterexample, which L^* uses to refine the learned state machine and make another conjecture. The process is guaranteed to terminate in time polynomial in the number of states and the size of the input alphabet. However, in practice, such an oracle is unavailable. The second approach is to generate random *sampling queries* and use those to test the equivalence between the conjecture and the black box. If a sampling query discovers a mismatch between a conjecture and the black box, refinement is done the same way as with the counterexamples that would be generated by equivalence queries. The sampling approach provides a probabilistic guarantee [4] on the accuracy of the inferred state machine. The third approach, called black box model checking [75], uses bounded model checking to compare the conjecture with the black box.

As discussed in Section 4.3, MACE requires an output message abstraction function $\alpha_O : \mathscr{M}_O \to \Sigma_O$, where $\mathscr{M}_O$ is the set of all concrete output messages, that abstracts concrete output messages into the abstract output alphabet. However, unlike the prior work [20], MACE requires no input abstraction function. We will extend the output abstraction function to sequences as follows. Let $o \in \mathscr{M}_O^*$ be a sequence of concrete output messages such that $|o| = n$. The abstraction of a sequence is defined as $\alpha_O(o) = \alpha_O(o_0) \cdots \alpha_O(o_{n-1})$.

4.4 Model-inference-Assisted Concolic Exploration

We begin this section by a high-level description of MACE, illustrated in Figure 4.2. After the high-level description, each section describes a major component of MACE: abstract model inference, concrete state-space exploration, and filtering of redundant concrete input messages together with the abstract model refinement.

A High-Level Description

Suppose we want to infer a complete Mealy machine $M = (Q, \Sigma_I, \Sigma_O, \phi, \lambda, q_0)$ representing some protocol, as implemented by the given program. We assume to know the output abstraction function α_O that abstracts concrete output messages into Σ_O. To bootstrap MACE, we also assume to have an initial set $\Sigma_{I0} \subseteq \Sigma_I$ of input messages, which can be extracted from either a regression test suite, collected by observing the communication of the analyzed program with the environment, or obtained from DART and similar approaches [34, 80, 17, 16]. The initial Σ_{I0} alphabet could be empty, but MACE would take longer to converge. In our work, we used regression test suites pro-

vided with the analyzed applications, or extracted messages from a single observed communication session if the test suite was not available.

Next, L^* infers the first state machine $M_0 = (Q_0, \Sigma_{I0}, \Sigma_O, \phi_0, \lambda_0, q_0^0)$ using Σ_{I0} and Σ_O as the abstract alphabets. In M_0, we find a shortest transfer sequence from q_0^0 to every state $q \in Q_0$. We use such sequences to drive the program to one of the concrete states represented by the abstract state q. Since each abstract state could correspond to a large cluster of concrete states (Fig. 4.1), we use dynamic symbolic execution to explore the clusters of concrete states around abstract states.

The state-space exploration generates sequences of concrete input and the corresponding output messages. Using the output abstraction function α_O, we can abstract the concrete output message sequences into sequences over Σ_O^*. However, we cannot abstract the concrete input messages into a subset of Σ_I, as we do not have the concrete input message abstraction function. Using all the concrete input messages for the L^*-based inference would be computationally infeasible. The state-space exploration discovers hundreds of thousands of concrete messages, because we run the exploration phase for hundreds of hours, and on average, it discovers several thousand new concrete messages per hour.

Thus, we need a way to filter out redundant messages and keep the ones that will allow L^* to discover new states. The filtering is done as follows. Suppose that s is a sequence of concrete input messages generated from the exploration phase and $o \in \Sigma_O^*$ a sequence of the corresponding abstract output messages. If there exists $t \in \Sigma_{I0}^*$ such that M_0 accepts t generating o, we discard s. Otherwise, at least one concrete message in the s sequence generates either a new state or a new transition, so we refine the input alphabet and compute $\Sigma_{I1} = \Sigma_{I0} \cup sup(s)$.

With the new abstract input alphabet Σ_{I1}, we infer a new, more refined, abstract model M_1 and repeat the process. If the number of messages is finite and either the exploration phase terminates or runs for a predetermined bounded amount of time, MACE terminates as well.

Model Inference with L^*

MACE learns the abstract model of the analyzed program by constructing sequences of input messages, sending them to the program, and reasoning about the responses. For the inference, we use Shahbaz and Groz's [82] variant of L^* for learning Mealy machines. The inference process is similar as in our prior work [20].

In every iteration of MACE, L^* infers a new state machine over Σ_{Ii} and the new messages discovered by the state-space exploration guided by M_i, and conjectures M_{i+1}, a refinement of M_i. Out of the three options for checking conjectures discussed in Section 4.3, we chose to check conjectures using the sampling approach. We could use sampling after each iteration, but we rather defer it until the whole process terminates. In other words, rather than doing sampling after each iteration, we use the subsequent MACE iterations instead of the traditional sampling. Once the process terminates, we generate sampling queries, but in no experiment we performed did sampling discover any new states.

The State-Space Exploration Phase

We use the model inferred in Section 4.4 to guide the state-space exploration. For every state $q^i \in Q_i$ of the just inferred abstract model M_i, we compute a shortest transfer sequence of input messages from the initial state q_0^i. Suppose the computed sequence is $s \in \Sigma_{Ii}^*$. With s, we drive the analyzed application to a concrete state abstracted by the q^i state in the abstract model. All messages $sup(s)$ are concrete messages either from the set of seed messages, or generated by previous state-space exploration iterations. Thus, the process of driving the analyzed application to the desired state consists of only computing a shortest path in M_i to the state, collecting the input messages along the path $q_0^i \xrightarrow{+} q^i$, and feeding that sequence of concrete messages into the application.

Once the application is in the desired state q^i, we run dynamic symbolic execution from that state to explore the surrounding concrete states (Figure 4.1). In other words, the transfer sequence of input messages produces a concrete run, which is then followed by symbolic execution that computes the corresponding path-condition. Once the path-condition is computed, dynamic symbolic execution resumes its normal exploration. We bound the time allotted to exploring the vicinity of every abstract state. In every iteration, we explore only the newly discovered states, i.e., $Q_i \backslash Q_{i-1}$. Re-exploring the same states over and over would be unproductive.

Thanks to the abstract model, MACE can easily compute the necessary input message permutations required to reach any abstract model state, just by computing a shortest path. On the other hand, approaches that combine concrete and symbolic execution have to negate multiple predicates and get the decision procedure to generate the required sequence of concrete input messages to get to a particular state. MACE has more control over this process, and our experimental results show that the increased control results in higher line coverage, deeper analysis, and more vulnerabilities found.

Model Refinement

The exploration phase described in Section 4.4 generates a large number (hundreds of thousands in our setting) of new concrete messages. Using all of them to refine the abstract model is both unrealistic, as inference is polynomial in the size of the alphabet, and redundant, as many messages are duplicates and belong to the same equivalence class. To reduce the number of input messages used for inference, Comparetti et al. [27] propose a message clustering technique, while we used a handcrafted an abstraction function in our prior work in Chapter 4. In this chapter, we take a different approach.

In the spirit of dynamic symbolic execution, the exploration phase solves the path-condition (using a decision procedure) to generate new concrete inputs, more precisely, sequences of concrete input messages. During the concrete part of the exploration phase, such sequences of input messages are executed concretely, which generates the corresponding sequence of output messages. We abstract the generated sequence of output messages using α_O. If the abstracted sequence can be generated by the current abstract model, we discard the sequence, otherwise we add all the corresponding concrete input messages to Σ_{Ii}. We define this process more formally:

$\hat{=}$[Filter Function] Let $\mathscr{M}_I$ (resp. $\mathscr{M}_O$) be a (possibly infinite) set of all possible concrete input (resp. output) messages. Let $s \in \mathscr{M}_I^*$ (resp. $o \in \mathscr{M}_O^*$) be a sequence of concrete input (resp. output) messages such that $|s| = |o|$. We assume that each input message s_j produces o_j as a response. Let $M_i \in \mathscr{A}$ be the abstract model inferred in the last iteration and $\mathscr{A}$ the universe of all possible Mealy machines. The filter function $f : \mathscr{A} \times \mathscr{M}_I^* \times \mathscr{M}_O^* \to 2^{\mathscr{M}_I}$ is defined as follows:

$$f(M_i, s, o) = \begin{cases} \emptyset & \text{if} \quad \exists t \in \Sigma_{Ii}^* \,.\, \lambda_i(t) = \alpha_O(o) \\ sup\,(s) & \text{otherwise} \end{cases}$$

In practice, a single input message could produce either no response or multiple output messages. In the first case, our implementation generates an artificial no-response message, and in the second case, it picks the first produced output message. A more advanced implementation could infer a subsequential transducer [90], instead of a finite-state machine. A subsequential transducer can transduce a single input into multiple output messages.

Once the exploration phase is done, we apply the filter function to all newly found input and output sequences s_j and o_j, and refine the alphabet Σ_{Ii} by adding the messages returned by the filter function. More precisely:

$$\Sigma_{I(i+1)} \leftarrow \Sigma_{Ii} \cup \bigcup_j f(M_i, s_j, o_j)$$

In the next iteration, L^* learns a new model M_{i+1}, a refinement of M_i, over the refined alphabet $\Sigma_{I(i+1)}$.

4.5 Implementation

In this section, we describe our implementation of MACE. The L^* component sends queries to and collects responses from the analyzed server, and thus can be seen as a client sending queries to the server and listening to the corresponding responses. Section 4.5 explains this interaction in more detail. Section 4.5 surveys the main model inference optimizations, including parallelization, caching, and filtering. Finally, Section 4.5 introduces our state-space exploration component, which is used as a baseline for the later provided experimental results.

L^* as a Client

Our implementation of L^* infers the protocol state machine over the concrete input and abstract output messages. As a client, L^* first resets the server, by clearing its environment variables and resetting it to the initial state, and then sends the concrete input message sequences directly to the server.

Servers have a large degree of freedom in how quickly they want to reply to the queries, which introduces non-deterministic latency that we want to avoid. For one server application we analyzed (Vino), we had to slightly modify the server code to assure synchronous response. We wrote

wrappers around the `poll` and `read` system calls that immediately respond to the L^*'s queries, modifying eight lines of code in Vino.

Model Inference Optimizations

We have implemented the L^* algorithm with distributed master-worker parallelization of queries. L^* runs in the master node, and distributes its queries among the worker nodes. The worker nodes compute the query responses, by sending the input sequences to the server, collecting and abstracting responses, and sending them back to L^*.

Since model refinement requires L^* to make repeated queries across iterations, we maintain a cache to avoid re-computing responses to the previously seen queries. L^* looks up the input in the cache before sending queries to worker nodes.

As L^*'s queries could trigger bugs in the server application, responses could be inconsistent. For example, if L^* emits two sequences of input messages, s and t, such that s is a prefix of t, then the response to s should be a prefix of the response to t. Before adding an input-output sequence pair to the cache, we check that all the prefixes are consistent with the newly added pair, and report a warning if they are inconsistent.

After each inference iteration, we analyze the state machine to find redundant messages (Definition 4.3) and discard them. This is a simple, but effective, optimization that reduces the load on the subsequent MACE iterations. This optimization is especially important for inferring the initial state machine from the seed inputs.

State-Space Exploration

Our implementation of the state-space exploration consists of two components: a shortest transfer sequence generator and the state-space explorer. A shortest transfer sequence generator is implemented through a simple modification of the L^* algorithm. The algorithm maintains a data structure (called observation table [4]) that contains a set of shortest transfer sequences, one for each inferred state. We modify the algorithm to output this set together with the final model. MACE uses sequences from the set to launch and initialize state-space explorers.

Our state-space explorer uses a combination of dynamic and symbolic execution [34, 80, 17, 16]. The implementation consists of a system emulator, an input generator, and a priority queue. The system emulator collects execution traces of the analyzed program with respect to given concrete inputs. Given a collected trace, the input generator performs symbolic execution along the traced path, computes the path-condition, modifies the path condition by negating predicates, and uses a decision procedure to solve the modified path condition and to generate new inputs that explore different execution paths. The generated inputs are then provided back to the system emulator and the exploration continues. We use the priority queue, like [35], to prioritize concrete traces that are used for symbolic execution. The traces that visit a larger number of new basic blocks, unexplored by the prior traces, have higher priority.

The system emulator provides the capability to save and restore program snapshots. To perform model-assisted exploration from a desired state in the model, we first set the program state to the

snapshot of the initial state. Then, we drive the program to the desired state using the corresponding shortest transfer sequence, and start dynamic symbolic execution from that state.

In all our experiments, we used the snapshot capability to skip the server boot process. More precisely, we boot the server, make a snapshot, and run all the experiments on the snapshot. We do not report the code executed during the boot in the line coverage results.

4.6 Evaluation

To evaluate MACE, we infer server-side models of two widely-deployed network protocols: Remote Framebuffer (RFB) and Server Message Block (SMB). The RFB protocol is widely used in remote desktop applications, including GNOME Vino and RealVNC[1]. Microsoft's SMB protocol provides file and printer sharing between Windows clients and servers. Although the SMB protocol is proprietary, it was reverse-engineered and re-implemented as an open-source system, called Samba. Samba allows interoperability between Windows and Unix/Linux-based systems. In our experiments, we use Vino 2.26.1 and Samba 3.3.4 as reference implementations to infer the protocol models of RFB and SMB respectively. We discuss the result of our model inference in Section 4.6.

Once we infer the protocol model from one reference implementation, we can use it to guide state-space exploration of other implementations of the same protocol. Using this approach, we analyze RealVNC 4.1.2 and Windows XP SMB, without re-inferring the protocol state machine.

MACE found a number of critical vulnerabilities, which we discuss in Section 4.6. In Section 4.6, we evaluate the effectiveness of MACE, by comparing it to the baseline state-space exploration component of MACE without guidance.

Experimental Setup

For our state-space exploration experiments, we used the DETER Security testbed [9] comprised of 3GHz Intel Xeon processors. For running L^* and the message filtering, we used a few slower 2.27GHz Intel Xeon machines. When comparing MACE against the baseline approach, we sum the inference and the state-space exploration time taken by MACE, and compare it to running the baseline approach for the same amount of time. This setup gives a slight advantage to the baseline approach because inference was done on slower machines, but our experiments still show MACE is significantly superior, in terms of achieved coverage, found vulnerabilities and exploration depth.

Model Inference and Refinement

We used MACE to iteratively infer and refine the protocol models of RFB and SMB, using Vino 2.26.1 and Samba 3.3.4 as reference implementations respectively. Table 4.1 shows the results of iterative model inference and refinement on Vino and Samba.

[1]Vino is the default remote desktop application in GNOME distributions; RealVNC reports over 100 million downloads (http://www.realvnc.com).

Program (Protocol)	Iter.	$\|Q\|$	$\|\Sigma_I\|$	$\|\Sigma_O\|$	Tot. Learning Time (min)
Vino	1st	7	8	7	142
(RFB)	2nd	7	12	8	8
Samba	1st	40	40	14	2028
(SMB)	2nd	84	54	24	1840
	3rd	84	55	25	307

Table 4.1: Model Inference Result at the End of Each Iteration. The second column identifies the inference iteration. The Q column denotes the number of states in the inferred model. The Σ_I (resp. Σ_O) column denotes the size of the input (resp. output) alphabet. The last column gives the total time (sum of all parallel jobs together) required for learning the model in each iteration, including the message filtering time. The learning process is incremental, so later iterations can take less time, as the older conjecture might need a small amount of refinement.

As discussed in Section 4.4, once MACE terminates, we check the final inferred model with sampling queries. We used 1000 random sampling queries composed of 40 input messages each, and tried to refine the state machine beyond what MACE inferred. The sampling did not discover any new state in any experiment we performed.

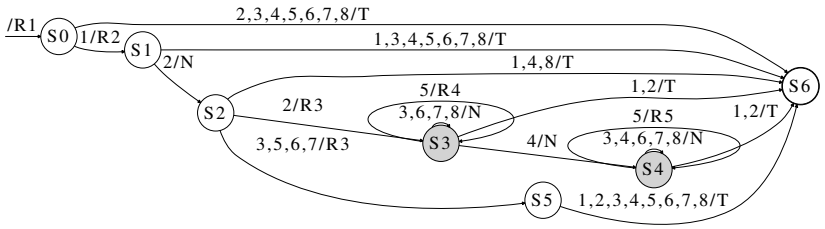
Vino. For Vino, we collected a 45-second network trace of a remote desktop session, using `krdc` (KDE Remote Desktop Connection) as the client. During this session, the Vino server received a total of 659 incoming packets, which were considered as seed messages. For abstracting the output messages, we used the message type and the encoding type of the outbound packets from the server. MACE inferred the initial model consisting of seven states, and filtered out all but 8 input and 7 output messages, as shown in Figure 4.3a.

Using the initial inferred RFB protocol model, the state-space explorer component of MACE discovered 4 new input messages and refined the model with new edges without adding new states (Figure 4.3b). We manually inspected the newly discovered output message (label R6 in Figure 4.3b) and found that it represents an outgoing message type not seen in the initial model.

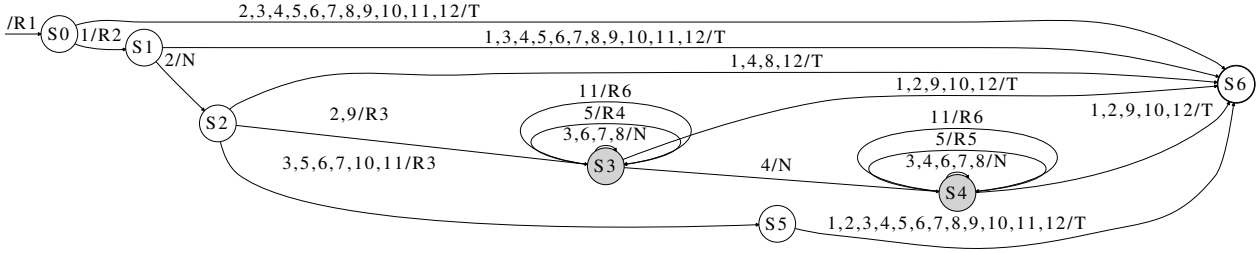
Since MACE found no new states that could be explored with the state-space explorer, the process terminated. Through manual comparison with the RFB protocol specification, we found that MACE has discovered all the input messages and all the states, except the states related to authentication and encryption, both of which we disabled in our experiments. Further, MACE found all the responses to client's queries[2].

We also performed an experiment with authentication enabled (encryption was still disabled). With this configuration, MACE discovered only three states, because it was not able to get past the checksum used during authentication, but discovered an infinite loop vulnerability that can be exploited for denial-of-service attacks. Due to space limits, we do not report the detailed results from this experiment, only detail the vulnerability found.

[2]There are two other output message types that are triggered by the server's GUI events and thus are outside of



(a) Original Vino's RFB Model Based on Observed Live Traffic.



(b) Final Vino's RFB Model Inferred by MACE.

Figure 4.3: Model Inference of Vino's RFB protocol. States in which MACE discovers vulnerabilities are shown in grey. The edge labels show the list of input messages and the corresponding output message separated by the '/' symbol. The explanations of the state and input and output message encodings are in Figure 4.4.

Label	Description
1	client's protocol version
2	byte 0x01 (securityType=None, clientInit)
3	setPixelFormat message
4	setEncodings message
5	frameBufferUpdateRequest message
6	keyEvent message
7	pointer event message
8	clientCutText message
9	byte 0x22
10	malformed client's protocol version
11	frameBufferUpdateRequest message with bpp=8 and true-color=false
12	malformed client's protocol version

(a) Input Legend.

Label	Description
R1	server's protocol version
R2	server's supported security types
R3	serverInit message
R4	framebufferUpdate message with default encoding
R5	framebufferUpdate message with alternative encoding
R6	setColourMapEntries message
N	no explicit reply from server
T	socket closed by server

(b) Output Legend.

Figure 4.4: Explanation of States and Input/Output Messages of the State Machine from Figure 4.3.

Label	Desc.	Label	Desc.	Label	Desc.	Label	Desc.	Label	Desc.
1	negprot	12	tconX	23	ntcreateX	34	ntcreateX	45	fclose
2	sesssetupX	13	nttrans	24	ntcreateX	35	mv	46	trans
3	sesssetupX	14	mkdir	25	trans2	36	trans2	47	tdis
4	tconX	15	invalid	26	trans2	37	openX	48	findnclose
5	unlink	16	rmdir	27	lockingX	38	trans2	49	dskattr
6	trans2	17	readX	28	writeX	39	setatr	50	findclose
7	trans2	18	lseek	29	checkpath	40	ntcreateX	51	exit
8	rmdir	19	close	30	mkdir	41	dskattr	52	dskattr
9	rmdir	20	ntrename	31	mv	42	fclose	53	ctemp
10	mkdir	21	openX	32	open	43	fclose	54	getatr
11	mkdir	22	mkdir	33	open	44	ulogoffX	55	create

(a) Input Legend.

Label	Desc.	Label	Desc.	Label	Desc.
R1	mkdir_success	R10	exit_success	R19	ulogoffX_success
R2	rmdir_success	R11	trans_success	R20	tconX_success
R3	open_success	R12	openX_success	R21	dskattr_success
R4	create_success	R13	trans2_success	R22	fclose_success
R5	mv_success	R14	findclose_success	R23	ntcreateX_success
R6	getatr_success	R15	findnclose_success	E	error
R7	setatr_success	R16	tdis_success	T	session terminated by server
R8	ctemp_success	R17	negprot_success		
R9	checkpath_success	R18	sesssetupX_success		

(b) Output Legend.

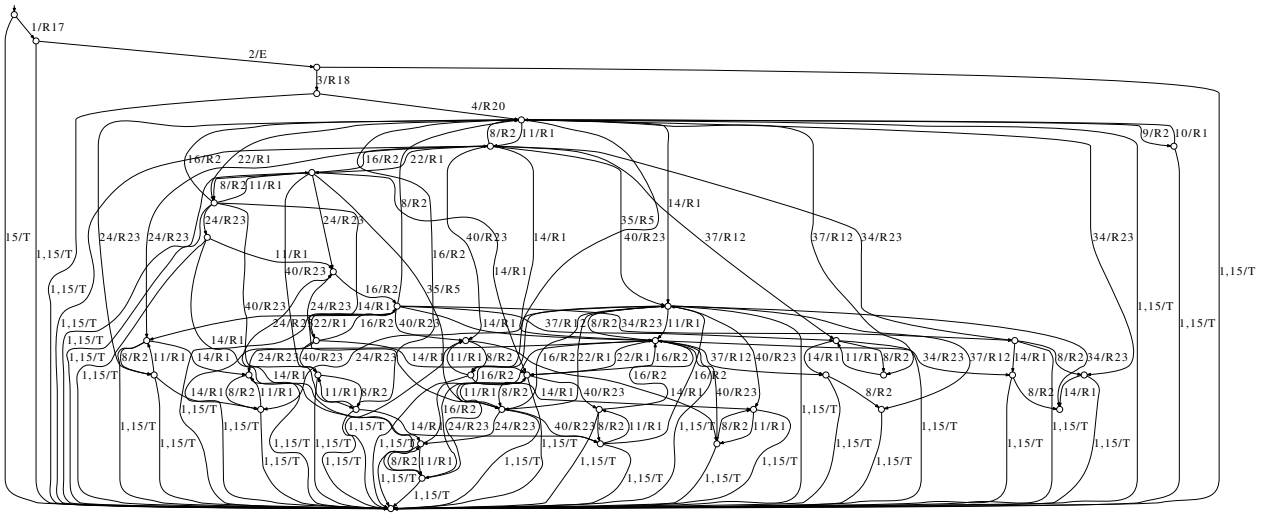
Figure 4.5: Explanation of Input/Output Messages of the State Machine from Figure 4.6.

Samba. For Samba, we collected a network trace of multiple SMB sessions, using Samba's `gentest` test suite[3], which generates random SMB operations for testing SMB servers. We used the default `gentest` configuration, with the default random number generator seeds. To abstract the outbound messages from the server, we used the SMB message type and status code fields; error messages were abstracted into a single error message type. The Samba server received a total of 115 input messages, from which MACE inferred an initial SMB model with 40 states, with 40 input and 14 output messages (after filtering out redundant messages). Figure 4.6a in Appendix shows the initial model.

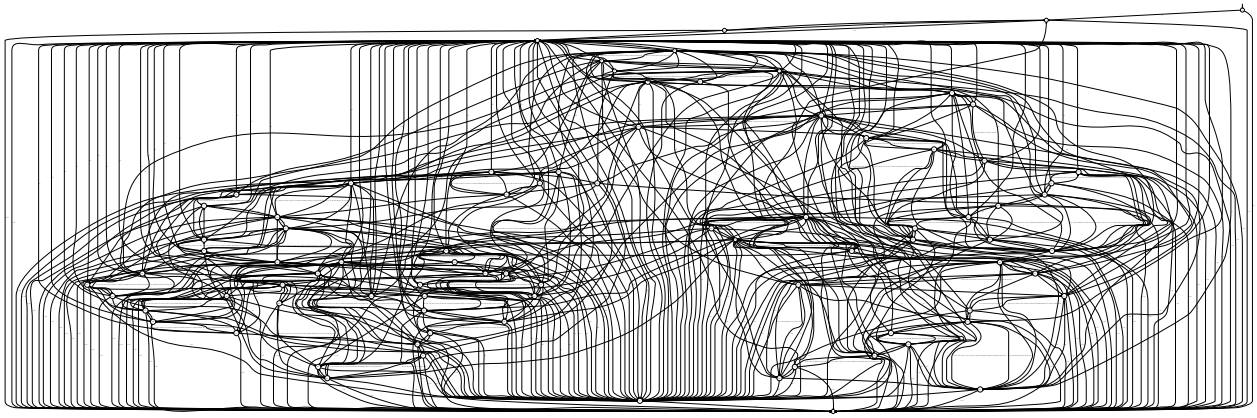
In the second iteration, MACE discovered 14 new input and 10 new output messages and refined the initial model from 40 states to 84 states. The model converged in the third iteration after adding a new input and a new output message without adding new states. Table 4.1 summarizes all three inference rounds. The converged model is depicted in Figure 4.6b.

our scope.

[3]http://samba.org/~tridge/samba_testing/



(a) The Initial SMB Model Inferred from the Seed Messages.



(b) Converged SMB Model.

Figure 4.6: The Inferred SMB Model from Samba.

Manually analyzing the inferred state machine, we found that some of the discovered input messages have the same type, but different parameters, and therefore have different effects on the server (and different roles in the protocol). MACE discovered all the 67 message types used in Samba, but the concrete messages generated by the decision procedure during the state-space exploration phase often had invalid message parameters, so the server would simply respond with an error. Such responses do not refine the model and are filtered out during model inference. In total, MACE was successful at pairing message types with parameters for 23 (out of 67) message types, which is an improvement of 10 message types over the test suite, which exercises only 13 different message types.

We identified several causes of incompleteness in message discovery. First, message validity is configuration dependent. For example, the `spoolopen`, `spoolwrite`, `spoolclose` and `spoolreturnqueue` message types need an attached printer to be deemed valid. Our experimental setup did not emulate the complete environment, precluding us from discovering some message types. Second, a single `echo` message type generated by MACE induced the server to behave inconsistently and we discarded it due to our determinism requirement. Although this is likely a bug in Samba, this behavior is not reliably reproducible. We exclude this potential bug from the vulnerability reports that we provide later. Third, our infrastructure is unable to analyze the system calls and other code executed in the kernel space. In effect, the computed symbolic constraints are underconstrained. Thus, some corner-cases, like a specific combination of the message type and parameter (e.g., a specific file name), might be difficult to generate. This is a general problem when the symbolic formula computed by symbolic execution is underconstrainted.

In our experiments, we used Samba's default configuration, in which encryption is disabled. The SMB protocol allows null-authentication sessions with empty password, similar to anonymous FTP. Thus, authentication posed no problems for MACE.

MACE converged relatively quickly in both Vino and Samba experiments (in three iterations or less). We attribute this mainly to the granularity of abstraction. A finer-grained model would require more rounds to infer. The granularity of abstraction is determined by the output abstraction function, (Section 4.3).

Discovered Vulnerabilities

We use the inferred models to guide the state-space exploration of implementations of the inferred protocol. After each inference iteration, we count the number of newly discovered states, generate shortest transfer sequences (Section 4.3) for those states, initialize the server with a shortest transfer sequence to the desired (newly discovered) state, and then run 2.5 hours of state-space exploration in parallel for each newly discovered state. The input messages discovered during those 2.5 hours of state-space exploration per state are then filtered and used for refining the model (Section 4.4). For the baseline dynamic symbolic execution without model guidance, we run $|Q|$ parallel jobs with different random seeds for each job for 15 hours, where $|Q|$ is the number of states in the final converged model inferred for the target protocol. Different random seeds are important, as they assure that each baseline job explores different trajectories within the program.

We rely upon the operating system runtime error detection to detect vulnerabilities, but other

detectors, like Valgrind[4], could be used as well. Once MACE detects a vulnerability, it generates an input sequence required for reproducing the problem. When analyzing Linux applications, MACE reports a vulnerability when any of the critical exceptions (`SIGILL`, `SIGTRAP`, `SIGBUS`, `SIGFPE`, and `SIGSEGV`) is detected. For Windows programs, a vulnerability is found when MACE traps a call to `ntdll.dll::KiUserExceptionDispatcher` and the value of the first function argument represents one of the critical exception codes.

MACE found a total of seven vulnerabilities in Vino 2.26.1, RealVNC 4.1.2, and Samba 3.3.4, within 2.5 hours of state-space exploration per state. In comparison, the baseline dynamic symbolic execution without model-guidance, found only one of those vulnerabilities (the least critical one), even when given the equivalent of 15 hours per state. Four of the vulnerabilities MACE found are new and also present in the latest version of the software at the time of writing. The list of vulnerabilities is shown in Table 5.1. The rest of this section provides a brief description of each vulnerability.

Vino. MACE found three vulnerabilities in Vino; all of them are new. The first one (CVE-2011-0904) is an out-of-bounds read from arbitrary memory locations. When a certain type of the RFB message is received, the Vino server parses the message and later uses two of the message value fields to compute an unsanitized array index to read from. A remote attacker can craft a malicious RFB message with a very large value for one of the fields and exploit a target host running Vino. The Gnome project labeled this vulnerability with the "Blocker" severity (bug 641802), which is the highest severity in their ranking system, meaning that it must be fixed in the next release. MACE found this vulnerability after 122 minutes of exploration per state, in the first iteration (when the inferred state machine has seven states, Table 4.1). The second vulnerability (CVE-2011-0905) is an out-of-bounds read due to a similar usage of unsanitized array indices; the Gnome project labeled this vulnerability (bug 641803) as "Critical", the second highest problem severity. This vulnerability is marked as a duplicate of CVE-2011-0904, for it can be fixed by patching the same point in the code. However, these two vulnerabilities are reached through different paths in the finite-state machine model and the out-of-bounds read happens in different functions. These two vulnerabilities are actually located in a library used by not only Vino, but also a few other programs. According to Debian security tracker[5], kdenetwork 4:3.5.10-2 is also vulnerable.

The third vulnerability (CVE-2011-0906) is an infinite loop, found in the configuration with authentication enabled. The problem appears when the Vino server receives an authentication input from the client larger than the authentication checksum length that it expects. When the authentication fails, the server closes the client connection, but leaves the remaining data in the input buffer queue. It also enters an deferred-authentication state where all subsequent data from the client is ignored. This causes an infinite loop where the server keeps receiving callbacks to process inputs that it does not process in deferred-authentication state. The server gets stuck in the infinite loop and stops responding, so we classify this vulnerability as a denial-of-service vulnerability. Unlike all other discovered vulnerabilities, we discovered this one when L^* hanged, rather than by catching signals or trapping the exception dispatcher. Currently, we have no way of detecting this

[4]http://valgrind.org/

[5]http://security-tracker.debian.org/tracker/CVE-2011-0904

Program	Vulnerability Type	Disclosure ID	Iter.	Jobs ($\|Q\|$)	Search Time			
					MACE		Baseline	
					per job (min)	total (hrs)	per job (min)	total (hrs)
Vino	Wild read (blocker)	*CVE-2011-0904*	1/2	7	122	15	>900	>105
	Out-of-bounds read	*CVE-2011-0905*	1/2	7	31	4	>900	>105
	Infinite loop	*CVE-2011-0906*†	1/2	7	1	1	N/A	N/A
Samba	Buffer overflow	CVE-2010-2063	1/3	84	88	124	>900	>1260
	Out-of-bounds read	CVE-2010-1642	1/3	84	10	14	>900	>1260
	Null-ptr dereference	Fixed w/o CVE	1/3	84	8	12	430	602
RealVNC	Out-of-bounds write	*CVE-2011-0907*	1/1	7	17	2	>900	>105
WinXP SMB	None	None	None	84	>150	>210	>900	>1260

Table 4.2: Description of the Found Vulnerabilities. The upper half of the table (Vino and Samba) contains results for the reference implementations from which the protocol model was inferred, while the bottom half (Real VNC and Win XP SMB) contains the results for the other implementations that were explored using the inferred model (from Vino and Samba). The disclosure column lists Common Vulnerabilities and Exposures (CVE) numbers assigned to vulnerabilities MACE found. The new vulnerabilities are *italicized*. The † symbol denotes a vulnerability that could not have been detected by the baseline approach, because it lacks a detector that would register non-termination. We found it with MACE, because it caused L^* to hang. The "Iter." column lists the iteration in which the vulnerability was found and the total number of iterations. The "Jobs" column contains the total number of parallel state-space exploration jobs. The number of jobs is equal to the number of states in the final converged inferred state machine. The baseline experiment was done with the same number of jobs running in parallel as the MACE experiment. The MACE column shows how much time passed before at least one parallel state-space exploration job reported the vulnerability and the total runtime (number of jobs $\times$ time to the first report) of all the jobs up to that point. The "Baseline" column shows runtimes for the baseline dynamic symbolic execution without model guidance. We set the timeout for the MACE experiment to 2.5 hours per job. The baseline approach found only one vulnerability, even when allowed to run for 15 hours (per job). The $> t$ entries mean that the vulnerability was not found within time t.

vulnerability with the baseline, so we do not report the baseline results for CVE-2011-0906.

Samba. MACE found 3 vulnerabilities in Samba. The first two vulnerabilities have been previously reported and are fixed in the latest version of Samba. One of them (CVE-2010-1642) is an out-of-bounds read caused by the usage of an unsanitized Security_Blob_Length field in SMB's Session_Setup_AndX message. The other (CVE-2010-2063) is caused by the usage of an unsanitized field in the "Extra byte parameters" part of an SMB Logoff_AndX message. The third one is a null pointer dereference caused by an unsanitized Byte_Count field in the Session_Setup_AndX request message of the SMB protocol. To the best of our knowledge, this vulnerability has never been publicly reported but has been fixed in the latest release of Samba. We did not know about any of these vulnerabilities prior to our experiments.

RealVNC. MACE found a new critical out-of-bounds write vulnerability in RealVNC. One

Program (Protocol)	Sequential Time (min)	Instruction Coverage			Total crashes (Unique crashes)	
		Baseline	MACE	improvement	Baseline	MACE
Vino (RFB)	1200	129762	138232	6.53%	0 (0)	2 (2)
Samba (SMB)	16775	66693	105946	58.86%	20 (1)	21 (5)
RealVNC (RFB)	1200	39300	47557	21.01%	0 (0)	7 (2)
Win XP (SMB)†	16775	90431	112820	24.76%	0 (0)	0 (0)

Table 4.3: Instruction Coverage Results. The table shows the instruction coverage (number of unique executed instruction addresses) of MACE after 2.5 hours of exploration per state in the final converged inferred state machine, and the baseline dynamic symbolic execution given the amount of time equivalent to (time MACE required for inferring the final state machine + number of states in the final state machine $\times$ 2.5 hours), shown in the second column. For example, from Table 4.1, we can see that Samba inference took the total of $2028+1840+307=4175$ minutes and produced an 84-state model. Thus, the baseline approach was given $84 \times 150 + 4175 = 16775$ minutes to run. The last two columns show the total number of crashes each approach found, and the number of unique crashes according to the location of the crash in parenthesis. Due to a limitation of our implementation of the state-space exploration (user-mode only), the baseline result for Windows XP SMB (marked †) was so abysmal, that comparing to the baseline would be unfair. Thus, we compute the Win XP SMB baseline coverage by running Samba's `gentest` test suite.

type of the RFB message processed by RealVNC contains a length field. The RealVNC server parses the message and uses the length field as an index to access the process memory without performing any sanitization, causing an out-of-bounds write.

Win XP SMB. The implementation of Win SMB is partially embedded into the kernel, and currently our dynamic symbolic execution system does not handle the kernel operating system mode. Thus, we were able to explore only the user-space components that participate in handling SMB requests. Further, we found that many involved components seem to serve multiple purposes, not only handling SMB requests, which makes their exploration more difficult. We found no vulnerabilities in Win XP SMB.

Comparison with the Baseline

We ran several experiments to illustrate the improvement of MACE over the baseline dynamic symbolic execution approach. First, we measured the instruction coverage of MACE on the analyzed programs and compared it against the baseline coverage. Second, we compared the number of crashes detected by MACE and by the baseline approach over the same amount of time. This number provides an indication of how diverse the execution paths discovered by each approach are: more crashes implies more diverse searched paths. Finally, we compared the effectiveness of MACE and the baseline approach to reach deep states in the final inferred model.

Instruction Coverage. In this experiment, we measured the numbers of unique instruction

addresses (i.e., EIP values) of the program binary and its libraries covered by MACE and the baseline approach. These numbers show how effective the approaches are at uncovering new code regions in the analyzed program. For Vino, RealVNC, and Samba, we used dynamic symbolic execution as the baseline approach and ran the experiment using the setup outlined in Section 4.6. We ran MACE allowing 2.5 hours of state-space exploration per each inferred state. To provide a fair comparison, we ran the baseline for the amount of time that is equal to the sum of the MACE's inference and state-space exploration times. As shown in Table 4.3, our result illustrates that MACE provides a significant improvement in the instruction coverage over dynamic symbolic execution.

As mentioned before, our tool currently works on user-space programs only. Because Windows SMB is mostly implemented as a part of the Windows kernel, the results of the baseline approach were abysmal. To avoid a straw man comparison, we chose to compare against Samba's `gentest` test suite, regularly used by Samba developers to test the SMB protocol. Using the test suite, we generate test sequences and measure the obtained coverage. As for other experiments, we allocated the same amount of time to both the test suite and MACE. The experimental results clearly show MACE's ability to augment test suites manually written by developers.

Number of Detected Crashes. Using the same setup as in the previous experiment, we measured the number of crashing input sequences generated by each approach. We report the number of crashes and the number of unique crash locations. From each category of unique crash locations, we manually processed the first four reported crashes. All the found vulnerabilities (Table 5.1) were found by processing the very first crash in each category. All the later crashes we processed were just variants of the first reported crash. MACE found 30 crashing input sequences with 9 of them having unique crash locations (the EIP of the crashed instruction). In comparison, the baseline approach only found 20 crashing input sequences, all of them having the same crash location.

Exploration Depth. Using the same setup as for the coverage experiment, we measured how effective each approach is in reaching deep states. The inferred state machine can be seen as a directed graph. Suppose we compute a spanning tree (e.g., [28]) of that graph. The root of the graph is at level zero. Its children are at level one, and so on. We measured the percentage of states reached at every level. Figure 4.7 clearly shows that MACE is superior to the baseline approach in reaching deep states in the inferred protocol.

4.7 Limitations

Completeness is a problem for any dynamic analysis technique. Accordingly, MACE cannot guarantee that all the protocol states will be discovered. Incompleteness stems from the following: (1) each state-space explorer instance runs for a bounded amount of time and some inputs may simply not be discovered before the timeout, (2) among multiple shortest transfer sequences to the same abstract state, MACE picks one, potentially missing further exploration of alternative paths, (3) similarly, among multiple concrete input messages with the same abstract behavior, MACE picks one and considers the rest redundant (Definition 4.3).

Our approach to model inference and refinement is not entirely automatic: the end users need

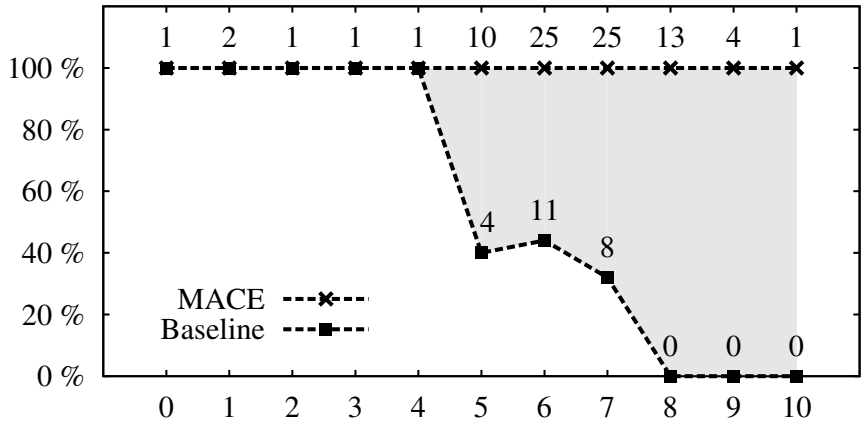


Figure 4.7: SMB Exploration Depth. The inferred state machine can be seen as a directed graph. Suppose we compute a spanning tree (e.g., [28]) of that graph. The root of the graph is at level zero. Its children are at level one, and so on. The figure shows the percentage of states visited at each level by MACE and the baseline approach. The numbers above points show the number of visited states at the given depth. The shaded area clearly shows that MACE is superior to the baseline approach in reaching deep states of the inferred protocol.

to provide an abstraction function that abstracts concrete output messages into an abstract alphabet. Coming up with a good output abstraction function can be a difficult task. If the provided abstraction is too fine-grained, model inference may be too expensive to compute or may not even converge. On the other hand, the inferred model may fail to distinguish two interesting states if the abstraction is too coarse-grained. Nevertheless, our approach provides an important improvement over our prior work [21], which requires abstraction functions for both input and output messages.

When using our approach to learn a model of a proprietary protocol, a certain level of protocol reverse-engineering is required prior to running MACE. First, we need a basic level of understanding of the protocol interface to be able to correctly replay input messages to the analyzed program. For example, this may require overwriting the cookie or session-id field of input messages so that the sequence appears indistinguishable from real inputs to the target program. Second, our approach requires an appropriate output abstraction, which in turn requires understanding of the output message formats. Message format reverse-engineering is an active area of research [29, 30, 15] out of the scope of this thesis.

Encryption is a difficult problem for every (existing) protocol inference technique. To circumvent the issue, we configure the analyzed programs not to use encryption. However, for proprietary protocols, such a configuration may not be available and techniques [14, 92] that automatically reverse-engineer message encryption are required.

4.8 Conclusions and Future Work

We have proposed MACE, a new approach to software state-space exploration. MACE iteratively infers and refines an abstract model of the protocol, as implemented by the program, and exploits the model to explore the program's state-space more effectively. By applying MACE to four server applications, we show that MACE (1) improves coverage up to 58.86%, (2) discovers significantly more vulnerabilities (seven vs. one), and (3) performs significantly deeper search than the baseline approach.

We believe that further research is needed along several directions. First, a deeper analysis of the correspondence of the inferred finite state models to the structure and state-space of the analyzed application could reveal how models could be used even more effectively than what we propose in this chapter. Second, it is an open question whether one could design effective automatic abstractions of the concrete input messages. The filtering function we propose in this chapter is clearly effective, but might drop important messages. Third, the finite-state models might not be expressive enough for all types of applications. For example, subsequential transducers [90] might be the next, slightly more expressive, representation that would enable us to model protocols more precisely, without significantly increasing the inference cost. Fourth, MACE currently does no white box analysis, besides dynamic symbolic execution for discovering new concrete input messages. MACE could also monitor the value of program variables, consider them as the input and the output of the analyzed program, and automatically learn the high-level model of the program's state-space. This extension would allow us to apply MACE to more general classes of programs.

Chapter 5

Compositional Bounded Model Checking for Real-world Programs

5.1 Introduction

Software Bounded Model Checking (BMC) is a powerful technique for finding bugs in bounded program executions. The technique constructs a formula, called a BMC *instance*, that encodes the behavior of a program up to a user-specified bound. BMC instances can capture bit-level operations and the memory model, and so generate precise information about bugs. However, existing BMC tools exhaust memory or time on programs containing a few thousand lines of code [56].

In this chapter, we ask: *is it possible solve a large instance by only solving instances that fit in memory*. Specifically, we seek to design a *compositional* BMC algorithm. An analysis technique is compositional if it can decompose a large problem into smaller problems, and solve only a small problem at a time. For example, Hoare logic is compositional because it allows for proving a property about a program by proving properties about its parts. Several static analysis techniques are compositional because they propagate information through a program one block at a time.

Decomposing BMC instances is challenging because dependencies between constraints are lost if the formula is decomposed. Imagine a procedure bar() which calls a procedure foo(**int** x) with argument 2. If the assertion is never violated, no formula that includes the behavior of both foo and bar will be satisfiable. If we decompose the problem and reason about bar() and foo(**int** x) separately, the assertion may be violated because there is no constraint on the input variable x. More generally, decomposing a BMC instance may render an unsatisfiable formula satisfiable because the context in which the code executes is not taken into account. We do not wish to report that an assertion *may be violated* because we lose the appealing property that BMC reports precise information about bugs.

Compositional Bounded Model Checking

We use two ideas to address the problems sketched above. First, we consider *preconditions for violation*, which are conditions under which a program is guaranteed to violate an assertion. Com-

puting such preconditions precisely would involve the expensive step of quantifier elimination while overapproximating preconditions leads to spurious bug reports. We compute the *underapproximate* preconditions for a violation, and weaken them iteratively. We use information from proofs generated by SAT solvers to *expand* the set of error states that must lead to a violation. As a result, we can decompose a BMC instance, consider multiple inputs to a piece of code, and preserve the accuracy of BMC with respect to bugs.

Our second idea is to incrementally generate BMC instances using information from proofs generated by SAT solvers. Solvers reason about semantic relationships between variables when they construct proofs, so proofs provide *relevance heuristics* [63] for tuning program analyses. By constructing BMC instances relevant to variables and operations appearing in a proof, we dramatically reduce the number of instances that must be considered. Our approach is closely related to the use of Craig interpolants in verification [66] and the formulae we construct also satisfy the interpolation criteria. Consequently, our technique is not only compositional, but like interpolation techniques, is property-driven.

Our tool BLITZ implements a compositional BMC technique that combines underapproximate preconditions with incremental construction of BMC instances. Both steps are guided by unsatisfiability proofs generated by SAT solvers. We have evaluated BLITZ on vulnerability benchmarks containing unmodified real-world programs and found that BLITZ faces no capacity problems and can find bugs in programs of approximately 100KLOC. In addition, we applied BLITZ to check critical Internet infrastructure software from the Internet Systems Consortium and found multiple new vulnerabilities.

Content and Contributions

We present an algorithm and tool for compositional BMC that boosts the capacity of BMC and can discover bugs in real-world programs with about 100KLOC. We make the following contributions.

1. We present a new BMC algorithm that achieves compositionality by computing underapproximate preconditions, and improves scalability by proof-guided incremental instance construction.

2. We demonstrate that the technique extends the state of the art of BMC by applying our tool BLITZ to real-world benchmarks containing up to about 100KLOC.

3. We deploy BLITZ on critical Internet infrastructure software and find multiple new vulnerabilities.

The chapter is organised as follows: We revisit the terminology of BMC in Section 5.3, where we also recall notions from program analysis. Our compositional BMC algorithm is presented in Section 5.4 and evaluated in Section 5.5. We discuss related work in Section 5.6 and conclude in Section 5.7.

```
1:  void main(unsigned char a)
2:  {
3:   char b = foo(a);
4:   char c = baz(b); /* some func */
5:   bar(b,c);
6:  }
7:  char foo(unsigned char a)
8:  {
9:   return a + 1;
10: }
```

```
11: void bar(unsigned char b, c)
12: {
13:  char d = b + 2;
14:  char e = c * 2;
15:  char f = qux(d,e); /* some func */
16:  if (e == 1)
17:   d = c;
18:  assert(d >= 'd');
19: }
```

Figure 5.1: An Example Program.

(a)

$b_1 = *;$
$c_1 = *;$
$d_1 = b_1 + 2;$
$e_1 = c_1 \times 2;$
$f_1 = *;$
$d_2 = (e_1 = 1)\ ?\ c_1\ :\ d_1;$
$d_2 <$ 'd'

(b)

$b_1 = a_0 + 1;$	$a_0 <$ 'a'
$c_1 = baz(b_1);$	–
$d_1 = b_1 + 2;$	$b_1 <$ 'b'
$e_1 = c_1 \times 2;$	$d_1 <$ 'd'
$f_1 = qux(d_1, e_1);$	–
$d_2 = (e_1 = 1)\ ?\ c_1\ :\ d_1;$	$d_1 <$ 'd' $\vee (e_1 = 1 \wedge c_1 <$ 'd')
$d_2 <$ 'd'	$d_2 <$ 'd'

Figure 5.2: Formulae generated by BLITZ (a) The BMC formula generated for bar. (b) The left column shows the formula generated for the program and the right column shows the preconditions computed if BLITZ is run at the granularity of statements. Statements that are bypassed are indicated by '–'.

5.2 Technique Overview

We present a run of BLITZ on a simple program. The example is intentionally simple to facilitate presentation.

Goal. The program in Figure 5.1 begins execution in main and contains calls to four procedures foo, baz, bar and qux. The procedures baz and qux have no side effects and are not shown. The goal is to determine if the assertion at line 18 of bar is violated. The standard BMC procedure will take as input a parameter k and construct an instance $\mathscr{M}_k$ in which loops and recursive procedures are unwound k times and procedure calls are inlined. This approach leads to large formulae.

Formula construction The first novelty of BLITZ is that we only generate formulae for small code fragments rather than the entire program. BLITZ begins by generating a formula for the behavior of bar as shown in Figure 5.2. Observe that the variables have been labelled with subscripts so that a variable has different indices as its value changes in the lifetime of the program. Techni-

cally the formula corresponds to Single Static Assignment (SSA) form [31]. The values of b and c, being inputs, are unknown and denoted $*$, as is the value of f, because the body of qux is not considered.

Underapproximate preconditions The formula for bar is satisfiable so a SAT solver can generate a satisfying assignment. This satisfying assignment may not correspond to a feasible execution because the calling context for bar does not appear in the formula. The second novelty of BLITZ is to use a satisfying assignment to first generate an unsatisfiable formula and then use a proof of unsatisfiability to obtain a *precondition for assertion violation*. In this case, BLITZ generates the precondition $(b_1 < \text{`b'})$, where 'b' denotes the ASCII value of the character b. This means that the assertion is violated if the value of the variable b_1 is strictly less than the ASCII value of the character b.

There are two differences between the precondition computed by BLITZ and that of standard techniques. It is common to compute weakest preconditions with respect to a correctness property. Instead, we compute preconditions with respect to an assertion violation. The second difference is that we do not necessarily compute weakest preconditions. The Dijkstra weakest precondition with respect to the assertion violation condition $(d < \text{`d'})$ is the formula

$$\begin{aligned}(2 \times c_1 = 1) &\implies (c_1 < \text{`d'}) \\ \wedge \neg(2 \times c_1 \neq 1) &\implies (b_1 + 2 < \text{`d'})\end{aligned}$$

which simplifies to

$$\begin{aligned}(2 \times c_1 = 1) &\implies (c_1 < \text{`d'}) \\ \wedge \neg(2 \times c_1 \neq 1) &\implies (b_1 < \text{`b'}).\end{aligned}$$

The precondition generated by BLITZ is an *underapproximation* of the weakest precondition. The practical benefit of our formula is that it is smaller and contains fewer Boolean operations, hence is easier for a solver to manipulate. This simplicity translates to significant performance benefits for larger formulae. The trade-off of this performance benefit is that BLITZ may miss bugs (produce false negatives) because it uses underapproximation. False negatives are mitigated by using a refinement strategy that weakens preconditions.

Incremental Instance Construction Next, BLITZ has to determine if bar can be called in a context satisfying $(b_1 < \text{`b'})$. Standard BMC and recent techniques like CORRAL [56] and ALTER [85] consider the callers and callees surrounding a procedure to generate such constraints. This approach would result in qux and baz being added to the instance.

BLITZ uses data-flow information in the program and only considers contexts that affect the underapproximate precondition. In the example, we can skip qux and baz and consider the call to foo at line 3. Solving an instance with the body of foo generates the precondition $(a_0 < \text{`a'})$, which is a sufficient precondition for the assertion violation. Solving this formula provides an input value to trigger the assertion violation. We have found that incrementally generating BMC instances by combining data-flow with underapproximate preconditions significantly reduces the number and size of instances we consider.

Fine-Grained Decomposition In the example above, we have argued at the level of procedures. Sometimes a procedure (or its unwinding) is too large to fit in memory. BLITZ supports decomposing BMC instances at different levels of granularity up to single statements. At the finest level, we start with a single statement directly preceding an assertion violation and incrementally compute necessary preconditions for single statements. We only work at this level of granularity when required or when configured by the user.

To complete the overview, Figure 5.2 (b) illustrates the formula constructed if we inline bar in main. Suppose BLITZ is run with a statement-level granularity, it will generate an underapproximate precondition for individual statements. Since data-flow is taken into account, statements not affecting the approximate preconditions are ignored.

5.3 Terminology and Preliminaries

We introduce background and notation about BMC and program analysis. The details of BMC included here are required later for correctness proofs and to clarify the differences between our technique and existing ones.

Syntax and Semantics of Programs

For simplicity of presentation, the formalisation here focuses on a small subset of C. Our technique and tool both apply to full ANSI C and are not subject to these restrictions.

Syntax Let *Var* be a set of variables. We use the symbol $*$ to denote a non-deterministic value. Let *Exp* be a set containing expressions in C and the $*$ symbol. Similarly, *BExp* is a set containing Boolean expressions in C and the $*$ symbol. A *statement* is an assignment, assumption, assertion, sequential composition of statements or a procedure call. The set of statements *Stmt* is defined inductively below.

$$\begin{aligned} \text{st} ::= \text{x} = \text{t} \mid \textbf{assume}(\text{b}) \mid \textbf{assert}(\text{b}) \mid \text{call P()} \\ \mid \text{st ; st} \mid \textbf{if}(\text{b})\,\text{st}\ \ \textbf{else}\ \ \text{st} \mid \textbf{while}(\text{b})\,\text{st} \end{aligned}$$

Semantics The semantics of programs is given by states and state transitions. A *program state* consists of the values of the program counter, global variables, and contents of the stack and the heap. Let *State* be the set of states.

The semantics of a program is given by a relation. A statement *st* defines a *transition relation* $T_{st} \subseteq State \times State$ that contains a pair (r,s) if executing *st* in the state *r* results in the state *s*. An assignment changes the value of a variable in a state and leaves all other states unchanged. The semantics of **assume**(bexp) is a relation that contains (s,s) if s satisfies bexp. The semantics of **assert**(bexp) is similar except that if s does not satisfy bexp there is a transition (s,e) to an error state e. The semantics of sequential composition p;q is the relational composition $T_p \circ T_q$. We write T_P to be the transition relation of a program P.

Error Reachability Program properties such as assertion violations can be formulated as reachability of states in a transition system. A state s is *reachable* in a program if there is an execution whose last state is s. The *error reachability problem* is to determine if an error state is reachable.

An error reachability technique is *sound* if whenever the technique reports an error, the error is reachable. An error reachability technique is *complete* if whenever the technique reports that the error is not reachable the error is indeed not reachable. Our definitions of soundness and completeness are given with respect to reachability and differ from the notions used in the correctness literature.

Bounded Model Checking

Bounded Model Checking (BMC) is a technique for finding bugs in bounded program executions. It operates by unwinding a program, translating the unwinding into a logical formula, and solving the formula.

Unwinding This material is based on [23]. A *k-unwinding* of a program P for a non-negative k is a program $unwind(P,k)$ defined inductively below.

$$\begin{aligned}
unwind(\mathrm{x = t}, k) &\triangleq \mathrm{x = t} \\
unwind(\textbf{assume}(\mathrm{b}), k) &\triangleq \textbf{assume}(\mathrm{b}) \\
unwind(\textbf{if}\,(\mathrm{b})\mathrm{p}\ \ \textbf{else}\ \ \mathrm{q}, k) &\triangleq \textbf{if}\,(\mathrm{b})\mathit{unwind}(\mathrm{p}, k) \\
&\qquad\qquad \textbf{else}\ \mathit{unwind}(\mathrm{q}, k) \\
unwind(\textbf{while}(\mathrm{b})\mathrm{p}, 0) &\triangleq \textbf{if}\,(\mathrm{b})\ \ \textbf{assume}(\mathrm{false}) \\
unwind(\textbf{while}(\mathrm{b})\mathrm{p}, k+1) &\triangleq \textbf{if}\,(\mathrm{b})\{\ \mathrm{p}; \\
&\qquad\qquad \mathit{unwind}(\textbf{while}(\mathrm{b})\mathrm{p}, k)\}
\end{aligned}$$

The definition of the unwinding for other C constructs is similar. Recursive procedures are handled by inlining. An *input variable* in a BMC unwinding is one that is never assigned. Input variables are the source of non-determinism in an unwound program.

Translation to Logic Statements in an unwinding are translated to formulae. An assignment x=x+1 (where = is assignment) cannot be written as a formula $x = x + 1$ (where $=$ represents mathematical equality) because x must have the same value on both sides of the equality in the formula. The statement x=x+1 is rewritten to x1 = x0 + 1 to make explicit that x may have different values before and after the assignment. This statement is then translated to a formula $x_1 = x_0 + 1$. More generally, a program is in Single Static Assignment (SSA) form if every variable is assigned exactly once in the program. Unwindings are translated to SSA form.

A k-unwinding of P in SSA form is translated into two logical formulae $\mathscr{B}(\mathrm{P})$ and $\mathscr{E}(\mathrm{P})$. The *behavioral constraint* $\mathscr{B}(\mathrm{P})$ encodes program executions. The *error constraint* $\mathscr{E}(\mathrm{P})$ encodes error conditions such as out-of-bounds array accesses, double frees, and assertion violations. If $\mathscr{B}$ implies $\neg\mathscr{E}$, no execution of a k-unwinding leads to an error. Conversely, a k-unwinding contains

an error if $\mathscr{B} \wedge \mathscr{E}$ is satisfiable. The satisfiability condition can be checked using a SAT or SMT [8] solver.

The behavior formula $\mathscr{B}(\mathrm{st})$ and error condition $\mathscr{E}(\mathrm{st})$ for a statement st as defined in [23] are recalled below. We write $form(\mathrm{t})$ for the logical expression corresponding to a C expression t.

st	$\mathscr{B}(\mathrm{st})$	$\mathscr{E}(\mathrm{st})$
x=t	$x = form(t)$	`true`
assume(b)	$form(b)$	`true`
assert(b)	`true`	$\neg form(b)$
P;Q	$\mathscr{B}(\mathrm{P}) \wedge \mathscr{B}(\mathrm{Q})$	$\mathscr{E}(\mathrm{P}) \wedge \mathscr{E}(\mathrm{Q})$

All statements except assertions modify the behavior constraint. The property is only modified by assertions. A BMC instance is *satisfiable* if $\mathscr{B} \wedge \mathscr{E}$ is satisfiable. A BMC instance is generated by traversing the control flow graph (CFG) of $unwind(P,k)$ in topological order starting from the initial vertex *init*. If x is an input variable of $unwind(P,k)$, we say that $form(x)$ is a *input variable* of of the BMC instance. We illustrate the construction of a BMC instance below.

Example 1. A program *P* is given on the left below with an unwinding on the right.

```
while(x < y)
  x = x + 1;
assert(x >= y);
```

$\longrightarrow$

```
if(x < y){
 x = x + 1;
 if(x < y){
  x = x + 1;
  if(x < y)
   assume(false);
 }
}
assert(x >= y);
```

The unwinding is translated into the formulae below.

$$\begin{aligned}\mathscr{B} \mathrel{\hat{=}} & \; x_1 = (x_0 < y_0)\ ?\ (x_0+1)\ :\ x_0 \\ & \wedge x_2 = (x_0 < y_0 \wedge x_1 < y_0)\ ?\ (x_1+1)\ :\ x_1 \\ & \wedge \neg(x_0 < y_0 \wedge x_1 < y_0 \wedge x_2 < y_0) \\ \mathscr{E} \mathrel{\hat{=}} & \; x_2 < y_0\end{aligned}$$

If the arithmetic and comparison operations and the memory model respect the semantics of C, the unwinding contains an assertion violation exactly if $\mathscr{B} \wedge \mathscr{E}$ is satisfiable.

Bit-vector logic is typically used to model numeric variables in C. Let $var(\varphi)$ be the set of variables in a formula φ. Let $\mathbb{B}$ be the set of values that variables can take. An *assignment* to a formula is a function $\sigma : var(\varphi) \to \mathbb{B}$.

Backward Reachability Computation

A classic approach to reasoning about programs is to approximate the set of reachable states. We recall reachability analysis because BLITZ combines BMC with ideas from reachability analysis.

A *symbolic encoding* is a representation of sets of states by data structures such as logical formulae. Let P be a program with a transition relation T, a set *Init* of initial states and a set *Err* of error states. We write $\bar{x}$ for a sequence of variables. The initial and error states are represented symbolically by the formulae $Init(\bar{x})$ and $Err(\bar{x})$. The transition relation generates a formula $T(\bar{x},\bar{y})$, in which $\bar{x}$ and $\bar{y}$ are of equal length. The *precondition transformer* defined below maps a formula $S(\bar{x})$ to a formula, called a *precondition* representing the predecessors of states represented by $S(\bar{x})$.

$$pre : Form \to Form \qquad\qquad pre(S(\bar{x})) \mathrel{\hat{=}} \exists \bar{x}.T(\bar{y},\bar{x}) \wedge S(\bar{x})$$

The set of *backward reachable states* $BReach(S(\bar{x}))$ contains those that lead to a state in $S(\bar{x})$. Below, we use the notation f^0 for the identity function and f^{i+1} for the function mapping x to $f(f^i(x))$. The backward reachable states have the well known characterisation below.

$$BReach(Err(\bar{x})) \iff \bigvee_{i\in\mathbb{N}} pre^i(Err(\bar{x}))$$

The equivalence suggests a naïve approach to computing states that lead to an error. The iterative computation terminates when a predicate that is a fixed point is reached. A *fixed point* of *Reach* is a predicate $F(\bar{x})$ satisfying the condition:

$$Reach(F(\bar{x})) \iff F(\bar{x})$$

A formula $Q(\bar{x})$ is an *underapproximate precondition* if $Q(\bar{x}) \implies pre(S(\bar{x}))$. The notions for overapproximations are similarly defined.

5.4 Compositional Bounded Model Checking

In this section we present an approach for finding deep bugs by combining BMC with approximate preconditions. An obstacle to scalability of BMC is that the size of a BMC instance grows beyond the capacity of solvers as the unwinding depth is increased. Reachability analysis with quantifier elimination faces similar computational obstacles. We combine BMC instances with underapproximate preconditions to scale BMC without producing false alarms about bugs.

Underapproximating Precondition Computation

We describe the use of SAT solvers to underapproximate preconditions with respect to a BMC instance and an error condition. Suppose we have a procedure P and a condition ψ that should hold after P executes. The formula ψ represents a set of states from which an error state can be reached. The set of predecessors $pre(\psi)$ represents all states that lead to a state in ψ through P. Computing

this set is expensive, so we compute an approximation. Program analysers typically overapproximate postconditions and may produce spurious results about bugs. We compute underapproximate preconditions to avoid spurious outcomes.

Our first observation is that the values of input variables in a satisfying assignment to a BMC instance dictate the value of other variables in the formula. Suppose a BMC instance $\mathscr{B}(\mathrm{P}) \wedge \mathscr{E}(\mathrm{P})$ has four variables w, x, y, z, with the input variables being x and y. A satisfying assignment of the form below can be viewed as a formula and this formula implies the sub-formula that ranges only over input variables (abbreviating `true` and `false` as `t` and `f` respectively).

$$(w{:}\mathsf{t}, x{:}\mathsf{f}, y{:}\mathsf{f}, z{:}\mathsf{f}) \rightsquigarrow w \wedge \neg x \wedge \neg y \wedge \neg z \qquad\qquad \Longrightarrow\ \neg x \wedge \neg y$$

Even though the sub-formula over input variables is weaker than the assignment, it is sufficient to constrain the values of all other variables, and hence is a *precondition for an error*. For the example above, the formula

$$\neg x \wedge \neg y \wedge \mathscr{B}(\mathrm{P}) \wedge \mathscr{E}(\mathrm{P})$$

has only one satisfying assignment because the values of x and y constraint the values of all other variables. The lemma below states this observation formally. We treat an assignment σ also as a formula and write $inp(\sigma)$ for the sub-formula of σ that contains only input variables.

Lemma 1. *If $\mathscr{B}(P) \wedge \mathscr{E}(P)$ is satisfied by an assignment σ, the formula $inp(\sigma) \wedge \mathscr{B}(P) \wedge \mathscr{E}(P)$ has a unique satisfying assignment. Consequently, the formula $inp(\sigma) \wedge \mathscr{B}(P) \wedge \neg\mathscr{E}(P)$ is unsatisfiable.*

Proof. Consider the satisfying assignment σ and the formula $\mathscr{B}(\mathrm{P})$. The semantics of $inp(\sigma) \wedge \mathscr{B}(\mathrm{P})$ is equivalent to replacing every input variable in $\mathscr{B}(\mathrm{P})$ by its truth value in σ. After such a replacement, only non-input variables remain, which by definition occur on the left-hand sides of assignment statements. Since the formula is generated by a program in SSA form, each non-input variable is assigned only once, and hence has a unique value for the whole formula. It follows that $inp(\sigma) \wedge \mathscr{B}(\mathrm{P}) \wedge \mathscr{E}(\mathrm{P})$ has a unique satisfying assignment.

For the second part, observe that σ satisfies $\mathscr{E}(\mathrm{P})$ and so does not satisfy $\neg\mathscr{E}(\mathrm{P})$. The constraints on input variables are determined by $\mathscr{B}(\mathrm{P})$ so $inp(\sigma) \wedge \mathscr{B}(\mathrm{P})$ must be uniquely satisfied by σ, so the conjunction $inp(\sigma) \wedge \mathscr{B}(\mathrm{P}) \wedge \neg\mathscr{E}(\mathrm{P})$ is unsatisfiable. □

The practical value of Lemma 1 is that we can derive an unsatisfiable formula from a satisfiable formula. A proof-generating SAT solver can generate a resolution refutation for a formula that is unsatisfiable. The refutation makes explicit what properties of the program are used to prove unsatisfiability and this information has numerous applications. Specifically, if an input variable does not appear in a refutation, the value of that variable does not affect the satisfiability of the restricted formula. We can use information from proofs to derive an underapproximation of a precondition that is more general than restricting an assignment to input variables.

To return to the example, suppose the constraint over input variables is $\neg x \wedge \neg y$ and the variable x does not occur in the refutation (with or without negation). We know that x is not required to

Algorithm 3: UNDERAPPROXIMATE PRECONDITION

```
approx-pre(F: a code fragment
    ψ: a postcondition for F
    σ: a satisfying assignment to 𝒷(F) ∧ ℰ(F) ∧ ψ
w: a weakening parameter)
    i := 0
    φ' := inp(σ, 𝒷(F))
    repeat
        φ := φ'
        (res, Π) := solve(φ ∧ 𝒷(F) ∧ ℰ(F) ∧ ¬ψ)
        φ' := generalise(φ, Π)
        i := i + 1
    until φ' = φ or i = w
    return φ
solve(φ: a formula)
    if φ is satisfiable then
        Let σ be a satisfying assignment
        return (sat, σ)
    else
        Let Π be a proof of unsatisfiability
        return (unsat, Π)
```

prove unsatisfiability of $\neg x \wedge \neg y \wedge \mathscr{B}(\mathrm{P}) \wedge \neg \mathscr{E}(\mathrm{P})$, so we can conclude that $\neg y \wedge \mathscr{B}(\mathrm{P}) \wedge \neg \mathscr{E}(\mathrm{P})$ is also unsatisfiable. Recall that if $A \wedge B$ is unsatisfiable, then A implies $\neg B$. In this example $\neg y \wedge \mathscr{B}(\mathrm{P})$ implies $\mathscr{E}(\mathrm{P})$, meaning that the set of states represented by $\neg y$ only leads to states that contain the error. Since $\neg x \wedge \neg y$ implies $\neg y$, we have generalised the precondition for the error.

The idea above is used to underapproximate preconditions. The reasoning we have sketched above does not require us to start with an assignment. It can be used to take a set of states that lead to an error and derive a larger set that only contains paths to an error. The restriction to states leading to an error is important to avoid spurious results about bugs.

The algorithm for underapproximate precondition computation is shown in Algorithm 3. We encapsulate interaction with the SAT solver by the method solve which takes as input a formula φ and returns a pair (res, θ), where res is the result of the satisfiability check. If φ is satisfiable, $\mathsf{res} = \mathsf{sat}$ and θ is a satisfying assignment. Otherwise, $\mathsf{res} = \mathsf{unsat}$ and θ is a proof of unsatisfiability.

The procedure approx-pre takes as input a program fragment F, a postcondition that F must satisfy in order to reach an error, a satisfying assignment σ to the formula $\mathscr{B}(\mathrm{F}) \wedge \mathscr{E}(\mathrm{F}) \wedge \psi$ and a parameter w. An *underapproximate precondition* is a formula ϕ satisfying the condition:

$$\phi \wedge \mathscr{B}(\mathrm{F}) \wedge \mathscr{E}(\mathrm{F}) \implies \psi$$

Every execution through F that starts in a state in ϕ leads to a state in ψ. The procedure approx-pre

returns an underapproximate precondition. The condition is approximate because it may not be the weakest such ϕ. Since ψ is a condition for reaching an error, the formula ϕ is a precondition for reaching an error.

The parameter w is used to weaken an underapproximate precondition. If w is 0, the underapproximate precondition is $inp(\sigma)$. If w is 1, the formula $inp(\sigma)$ is weakened using information from an unsatisfiability proof. We have considered two variants for the procedure generalise. One variant is to weaken ϕ by removing all variables that do not occur in the proof Π. To do this, we first walk the proof backwards to eliminate unnecessary deductions. Further techniques for computing minimal unsatisfiable cores of unsatisfiable formulae can also be used [61].

Lemma 2. *The formula* approx-pre(P, ψ, σ, w) *is an underapproximate precondition of P with respect to* ψ.

Proof. The invariant of the loop in the algorithm is that $\phi \wedge \mathscr{B}(\mathrm{F}) \wedge \mathscr{E}(\mathrm{F}) \wedge \neg\psi$ is unsatisfiable. This is because the generalisation step only reduces an unsatisfiable formula to a smaller unsatisfiable formula and projects out the portion of ϕ in this smaller formula. It follows that $\phi \wedge \mathscr{B}(\mathrm{F}) \wedge \mathscr{E}(\mathrm{F})$ implies ψ, so ϕ is a precondition of ψ. □

Incremental, Data-Flow-Based Instance Construction

If a BMC instance is satisfiable, we use approx-pre to construct an underapproximate precondition. To determine if there is indeed a bug, we need to propagate the precondition to the entry point of the program. The standard approach to propagate facts through a program is to use the Control Flow Graph (CFG). Propagation is wasteful if information is propagated through code not relevant for reaching an error.

We use the term *fragment* for a piece of code that is either a procedure or a configurable number of sequential statements. BMC instances are generated incrementally using a granularity specified by the user. A code fragment F *defines* a variable x if x occurs on the left-hand side of an assignment in F and *uses* the variable if the variable occurs in an expression. In program analysis, a *use-def* graph makes data-flow explicit.

We combine both data- and control-flow information to prune the number of instances that are generated. Given a condition for an error, we only use fragments that define variables in that condition to generate BMC instances. We show in the experiments section that this seemingly simple optimisation has significant impact on the number of instances generated, and consequently on the size of programs that can be analysed.

Instances are incrementally constructed using Algorithm 4. The procedure prev-code takes as input a fragment F and a parameter d which specifies the granularity of fragment. It returns a set of fragments that execute before F. At the granularity of procedures, we return the nodes before F in the call graph of the program. At the granularity of statements, we return statements preceding F in the CFG.

The procedure next-inst takes as input a fragment F that has already been analysed, a formula ψ, which is a precondition for F to reach an error, and a decomposition parameter d, which specifies the granularity of fragments. It returns a set of fragments that precede F and which define variables

Algorithm 4: ADJUSTABLE, INCREMENTAL BMC INSTANCE CONSTRUCTION

```
next-inst(F: a code fragment
    ψ: a condition for error
d: a decomposition parameter)
    Frag := prev-code(F, d)
    nObj := ∅
    repeat
        foreach fragment G in Frag do
            Remove G from Frag
            if G defines variables in ψ then
                Add G to nObj
            else
                Frag := Frag ∪ prev-code(G, d)
    until Frag = ∅
    return nObj
prev-code(F: a code fragment, d: a decomposition parameter)
    Frag := ∅
    if d = proc then
        Add callers of F in the call graph to Frag
    else if d = stmt then
        Add statements before F in the CFG to Frag
    return Frag
```

in ψ. It is implemented by iteratively calling prev-code until only fragments manipulating variables in ψ are obtained.

The BLITZ Algorithm

BLITZ combines underapproximate precondition computation with incremental construction of BMC instances to decompose the problem of finding a bug in a large instance to that of solving a sequence of smaller BMC instances. The advantage is that every instance fits in memory and that a smaller number of instances is considered than generating instances based on control flow. The algorithm is parameterised, so that the user may fine-tune the settings used for decomposition and precondition computation using domain-specific knowledge or data from benchmarks.

The procedure BLITZ in Algorithm 5 takes as input a program P which contains assertions, an unwinding bound k for loops and recursive procedures, a weakening parameter w for precondition approximation, and a decomposition granularity d. The algorithm returns an input vector $\bar{v}$ if there exists an execution of P with values specified by $\bar{v}$ that leads to an assertion violation. If no violation is found it may be because no error exists in k unwindings, or because relevant states were missed by the underapproximation.

The main data-structure in BLITZ is a set of *obligations Obj*. An *obligation* contains a fragment

F and a formula ψ representing a condition for an error. The initial value of ψ is the set of assertion violations possible in a program. BLITZ proceeds by solving the instance generated by each fragment in *Obj*. In addition to the behavior and error constraint, we also use the precondition ψ as an error constraint.

If a BMC instance is unsatisfiable, it is eliminated from *Obj* because its behavior is no longer relevant for finding a bug. Otherwise, the instance could be satisfiable either because the error is reachable or because the context in which the fragment F executes is not taken into account. We first underapproximate the error precondition of F using the method given earlier and then propagate this precondition to other fragments in the program. The subsequent fragments are determined by following the control- and data-flow of the program.

There are two outcomes which make BLITZ terminate. If the entry point of the program is reached and the precondition obtained is satisfiable, a satisfying assignment σ defines an input vector $inp(\sigma)$ which can be used to trigger an error. The other outcome is an empty set of obligations. This can occur either if no assertion is violated, or if the underapproximations have lost information about states leading to an error. In the latter case, there is nothing conclusive to report.

The precondition weakening approach proposed in Algorithm 3 can be viewed as an *eager* weakening approach that eagerly weakens each precondition up to a pre-determined bound w. We extend BLITZ with a *lazy* approach that avoids having to guess the appropriate weakening bound. The lazy approach starts by weakening each precondition once. If the entry of the program is reached with inconclusive results, BLITZ increments w and backtracks to further weaken a previously computed precondition. The weakening sites are chosen based on data-flow and only applied if a precondition is not the weakest precondition.

Theorem 2. *If* $\mathsf{BLITZ}(P,k,w,d)$ *returns an input vector, an assertion violation is reachable in P.*

5.5 Implementation and Evaluation

Implementation and Comparison to Other Tools

We implemented BLITZ within the CProver framework, which is the basis of the bounded model checker CBMC, which targets ANSI-C [23]. To extract proofs of unsatisfiability, we use a proof-logging version of the SAT solver MINISAT [33].

Our implementation uses several practical optimisations to improve upon the compositional BMC algorithm we presented. For one, we interleave unwinding and formula generation, to avoid storing large structures in memory. This optimisation influences the architecture of implementation. We try to avoid recomputing information about procedures whenever possible. When a procedure has different callees, we also try to reuse preconditions that were computed earlier, if relevant. This optimisation is a rather weak form of procedure summarisation used in program analysis.

We compare BLITZ to other tools at a conceptual level. Tools with the similar goal of bug-finding and which we compare with are CBMC, ESBMC and CORRAL. Architecturally, CBMC and ESBMC are similar — both tools inline all procedures into a single BMC instance. Like BLITZ,

Algorithm 5: THE BLITZ ALGORITHM

```
BLITZ(P: a program,
      k: a bound for BMC
      w: a weakening parameter
  d: a decomposition granularity)
    Obj := ∅ //BMC obligations

    foreach fragment F containing an assertion do
      Add (F, ℰ(F)) to Obj
    repeat
      Remove (F, ψ) from Obj
      (res, σ) := solve(ℬ(F) ∧ ℰ(F) ∧ ψ)
      if F is the program entry point then
        return "Violation triggered by input inp(σ)"
      else if res = sat then
        φ := approx-pre(F, ψ, σ, w)
        Obj := Obj ∪ next-inst(F, φ, d)
    until Obj is empty
```

CORRAL combines bounded model checking with nondeterminism to bound the scope of analysis, and is competitive with the state-of-the-art [56]. Unlike BLITZ, CORRAL inlines procedures on demand, instead of propagating preconditions. For solvers, CORRAL and ESbmc use Z3 [70], an efficient SMT solver; CBMC and BLITZ use the CProver framework, which directly reduces a BMC instance to SAT and solves it with a SAT solver. Section 5.6 contains a more detailed discussion of the algorithmic content of these tools.

Benchmarks

We evaluate BLITZ on the Bugbench data set to measure its performance in detecting known vulnerabilities. Bugbench is an independent benchmark suite for systematic evaluation of bug detection tools [59]. It aims to be a representative collection of real-world C programs with known bugs. In our evaluation, we used the vulnerability subset of the suite that focuses on memory safety violations, the most critical type of bugs. Our results are in Section 5.5.

We also applied BLITZ to production software from the Internet Systems Consortium to search for unknown vulnerabilities. The ISC develops and distributes critical Internet infrastructure software to Internet providers worldwide[1]. We found new vulnerabilities in multiple production releases of ISC programs, currently deployed in the Internet. We discuss these results in Section 5.5.

In Section 5.5, we evaluate BLITZ under configurations regarding propagation (control vs data-flow) and precondition weakening. We find that data-flow based propagation is key to BLITZ's

[1]ISC software is deployed by over 80% of Internet providers worldwide (http://www.isc.org/).

scalability. We also find that the lazy approach to weakening preconditions works better than the eager approach.

All experiments were performed on Intel Xeon 2.93 GHz machines with 32GB RAM.

Evaluation on the Bugbench Vulnerability Benchmarks

The vulnerability benchmarks in Bugbench consist of 7 programs `polymorph`, `ncompress`, `man`, `gzip`, `bc`, `squid` and `cvs`. All benchmarks are labelled with the location and type of vulnerability: stack overflow, global overflow, heap overflow and double free. We list these programs in Table 5.1 with the labels B1-8, and indicate their sizes and type of vulnerability they contain.

Into each program, we inserted an assertion that if violated would trigger the vulnerability. We also statically determined the minimum unwinding k required to reach the vulnerability. This value is shown in the table and the same value was used with all tools. We then used CBMC, ESBMC, CORRAL and BLITZ to check for assertion violations. CBMC and ESBMC support slicing, which reduces the size of BMC instances. We ran these tools with slicing turned on. All tools terminate once an assertion violation is found, or report no violation on termination. We used a timeout of 24 hours (86400s).

As listed in the table, an unwinding bound of 1-3 was sufficient to find most vulnerabilities. The only exception was B2, where an unwinding of $k = 2048$ was required to reach a buffer overflow. This was the number of iterations to traverse a buffer of that size. We ran BLITZ with data-flow driven propagation and lazy weakening of preconditions. We also fixed the granularity of each analyzed code fragment in BLITZ at the level of a procedure in all our experiments. Thus, each analyzed BMC instance corresponds to a single procedure.

L.	Program	KLOC	Vulnerability	k	CBMC			CORRAL			BLITZ				
					Time (s)	Mem. (MB)	Bug	Time (s)	Mem. (MB)	Bug	#W	#R	Time (s)	Mem. (MB)	Bug
Known Vulnerabilities — BugBench Benchmark															
B1	poly	0.7	Stack Overflow	1	1	21	✓	2	10	✓	1	2	1	18	✓
B2	poly	0.7	Global Overflow	2048	2427	>32768	✗	>86400	8828	✗	1	3	959	10165	✓
B3	ncomp	1.9	Stack Overflow	1	12	258	✓	6	32	✓	1	3	6	506	✓
B4	man	4.7	Global Overflow	2	1692	>32768	✗	16	119	✓	1	3	8	156	✓
B5	gzip	8.2	Global Overflow	1	1567	>32768	✗	20	384	✓	1	3	5	67	✓
B6	bc	17.0	Heap Overflow	1	4733	>32768	✗	>86400	212	✗	11	10	258	2563	✓
B7	squid	93.5	Heap Overflow	1	>86400	15462	✗	>86400	22715	✗	1	4	1664	9024	✓
B8	cvs	114.5	Double Free	3	>86400	4301	✗	203	56	✗	1	30	2128	3871	✓
Discovered New Vulnerabilities — ISC Programs															
I1	aftr	13.3	Null Ptr Deref.	1	664	>32768	✗	85	482	✓	1	2	74	959	✓
I2	sntp	42.1	Null Ptr Deref.	1	5479	>32768	✗	25	136	✓	2	3	14	227	✓

Table 5.1: Comparison of CBMC, CORRAL and BLITZ on Benchmarks. Rows B1-8 contain the evaluation results on known vulnerabilities in the Bugbench benchmark, while rows I1-2 contain the evaluation results on programs from the Internet Systems Consortium (ISC), where BLITZ and CORRAL discovered new vulnerabilities. The "L." column lists the benchmark labels. The "KLOC" column indicates the size of each benchmark program in thousands of lines of code. The "k" column lists the (statically determined) minimum number of (loop and recursion) unwindings required to trigger the violation condition in each benchmark. The "Time(s)" column indicates the length of time for which the tool ran with >86400 indicating that the tool exceeded the 24-hour timeout. The "Mem.(MB)" column shows the maximum amount of memory used by the tool in MB. We write >32768 to indicate that the tool exceeded 32GB of memory. The "Bug" column indicates whether the vulnerability was found by the tool on termination. The "#W" column indicates the number of precondition *weakening* passes used in BLITZ, detected automatically using the lazy approach. The "#R" column shows the number of procedure *refinements* in the propagation chain that found the bug.

The evaluation results are in rows B1-8 of Table 5.1. CBMC and ESBMC produced similar results, so we omit ESBMC from Table 5.1 to conserve space.

All tools find a vulnerability in benchmarks B1, B3, B4 and B5, with BLITZ requiring the least amount of time. BLITZ is the only tool that does not time out and finds vulnerabilities in B2, B6, B7 and B8. The maximum time required by BLITZ was 35.47 minutes to find a double-free vulnerability in `cvs`. The time and memory required by BLITZ increases with the size of the benchmark and unwinding bound but we did not observe a timeout or memory exhaustion.

The experiments demonstrate the utility of underapproximate preconditions for bug-finding. The underapproximate preconditions did not usually require more than one weakening, with the exception of B6, where BLITZ backtracked to weaken the precondition 11 times. The results suggest that our compositional BMC approach is effective for finding violations on real-world programs of significant sizes.

CBMC and BLITZ have similar performance on the smallest benchmark (B1). Both tools also found the bug in B3. On benchmarks B2, B4, B5 and B6, CBMC runs out of memory memory, and on benchmarks B7 and B8, CBMC exceeded the 24-hr timeout.

CORRAL has performance similar to BLITZ on benchmarks B1, B3, B4 and B5, and outperforms CBMC on benchmark B3. However, CORRAL exceeded the 24-hr timeout on benchmarks B2, B6 and B7, and terminated prematurely without finding the vulnerability on the largest benchmark B8.

Evaluation on ISC Programs

We now evaluate BLITZ on open-source production software from the Internet Systems Consortium (ISC). We use the current production release of `aftr v1.1` and `sntp v4.2.6p5`. The Address Family Translation Router (`aftr`) plays a central role in transitioning the global Internet from IPv4 to IPv6 — it provides backwards compatibility to transport IPv4 traffic over IPv6 carrier infrastructure. The transitioning process was launched on June 6, 2012[2] and is currently ongoing, using the version of `aftr` in this evaluation. The program `sntp` is a reference implementation of the Simple Network Time Protocol that provides clock synchronisation over the Internet.

We focus on finding Null pointer dereference vulnerabilities, which are not represented in Bugbench. To check for Null pointer dereferences, we use the default Null pointer dereference assertions inserted by CProver. Since we have no knowledge of any vulnerability in the ISC programs, we used an unwinding depth of $k = 1$ for all the tools.

The results are rows I1-2 of Table 5.1. Again, ESbmc produces similar results to CBMC. CBMC quickly ran out of memory on these programs. CORRAL and BLITZ separately found the same vulnerabilities within minutes. We manually verified the results and have confirmed these vulnerabilities with the ISC, who will be patching their software for redistribution to affected Internet providers worldwide.

We briefly describe the two vulnerabilities in the following paragraphs.

[2]The Internet Society declared June 6, 2012 the World IPv6 Launch day.

`aftr`**.** A pointer `ss1` is dereferenced after a function call `ss1 = stdio_open()`. The function `stdio_open()` calls `fileno(stdin)` in its body, and returns `NULL` if `fileno(stdin)` returns -1 on stream failure. An attacker with control over the program's environment `stdin` will be able to exploit the Null pointer dereference.

`sntp`**.** A pointer `bcastaddr` is dereferenced after being allocated a buffer through a call to `realloc`. However, `realloc` could return `NULL` when memory allocation fails. An attacker who is able induce memory allocation failure will be able to exploit the Null pointer dereference.

Evaluation of BLITZ Features

We now evaluate BLITZ using different configurations for propagation and precondition weakening. First, we compare the effect of using data-flow for propagating preconditions to that using control-flow on performance. Second, we compare the performance of an eager and a lazy weakening strategy. In eager weakening, preconditions are weakened several times before propagation, while in lazy weakening, all propagation is performed first and preconditions are weakened only if an inconclusive result is obtained at the entry of the program. The timeout for the experiments was 24 hours (86400s).

Data vs Control-flow Propagation Fig. 5.3 compares the running time between using precondition-guided data-flow propagation and control-flow based propagation. In 6 of 10 benchmarks, control-flow based propagation leads to a timeout. Barring one benchmark on which the two techniques perform equally, data-flow based propagation is always superior.

The results suggest data-flow based propagation is key to scalability of BLITZ. Existing tools such as CORRAL [56] and ALTER [85] follow control-flow and inline procedures on demand. Using information in preconditions to construct instances allows BLITZ to sidestep code that is irrelevant for detecting a violation. For example, in one of the largest programs (`squid`), a single call to the procedure `comm_close` would have required the analysis of up to 715 million unique procedure instances if control-flow-based inlining was used. BLITZ however did not analyze the `comm_close` procedure.

Lazy vs Eager Weakening We compare lazy and eager weakening of underapproximate preconditions. A lazy approach starts the analysis using a weakening bound 1. When BLITZ terminates with an inconclusive result, the weakening bound is incremented, and BLITZ backtracks to the first computed precondition on the data-flow path that is not equivalent to the weakest precondition. We configured the eager approach to use 11 passes, since this was the upper bound detected by the lazy approach (see Table 5.1).

Fig. 5.4 compares the effect of lazy and eager weakening on running time. The lazy approach leads to shorter running times because weakening a precondition twice suffices to discover a bug in most benchmarks.

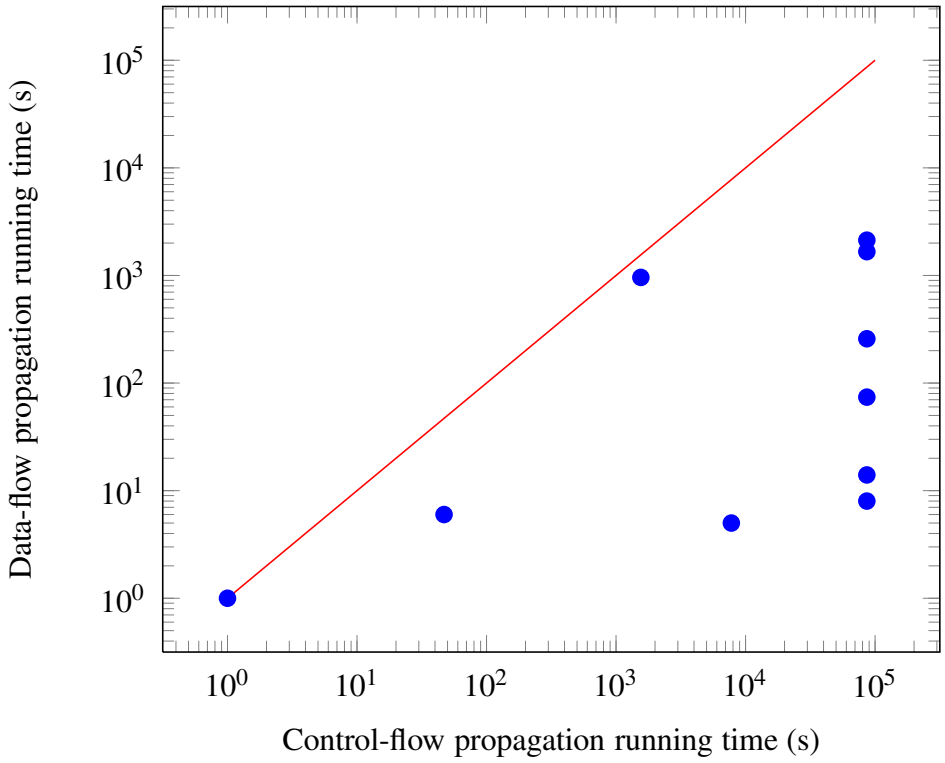


Figure 5.3: A comparison of BLITZ running times with data-flow against control-flow propagation, with 24-hr timeout.

5.6 Related Work

This thesis lies at the intersection of model checking and program analysis. Of several techniques to automatically compute information about a program, SLAM is notable for its success as a software model checker [6]. SLAM requires repeated calls to a theorem prover to construct abstractions and refines the abstraction using counterexamples. One refinement strategy uses weakest preconditions, but these are computed over single paths, unlike program fragments as in our case.

The compositional approach to system analysis is key for overcoming the complexity of any real-world system. We focused on composition defined with respect to sequential composition and procedure calls, as in Hoare logic [47]. Compositionality is exploited in program analysis [36] and symbolic execution [3], but we are not aware of it being used in BMC to date. The BLITZ algorithm is composition in the same sense, but differs from these above approaches because it uses data-flow to further reduce the decomposition of the program.

A key technique for relevance-driven program analysis is Craig interpolation, which has numerous applications in formal verification. Interpolation was applied to over-approximate reachable states in hardware model checking in [64], to infer preconditions in [65, 54], and to learn annotations from failed explorations to improve backtracking efficiency [66, 85]. We do not use

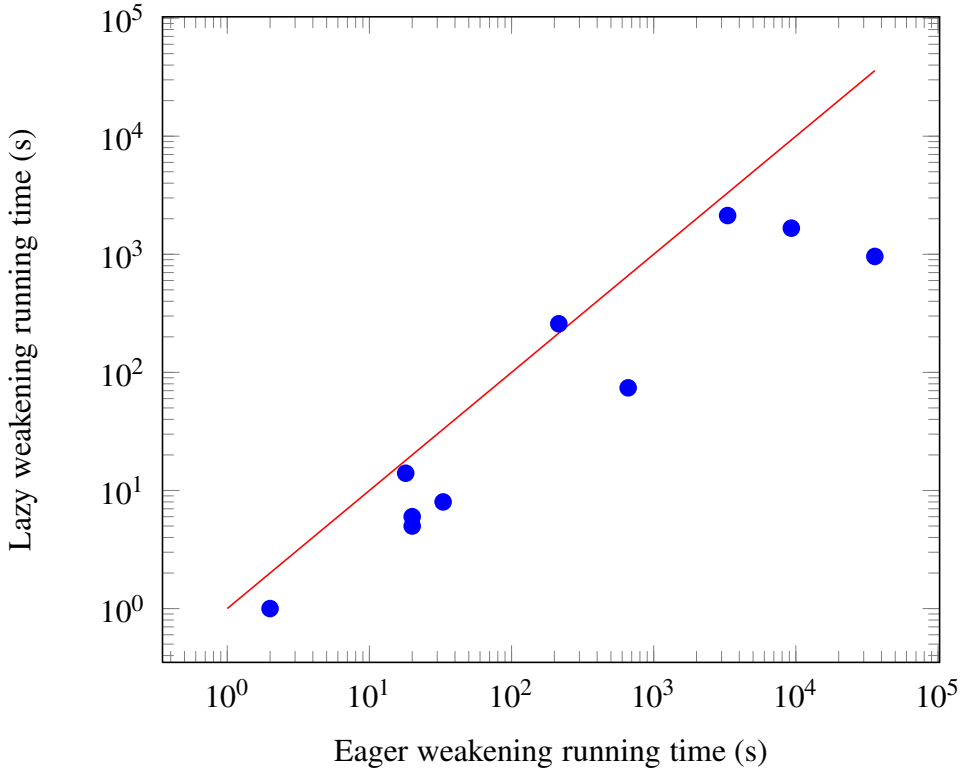


Figure 5.4: A comparison of BLITZ running times with lazy against eager weakening of preconditions, with 24-hr timeout.

standard techniques for computing interpolants from proofs because these lead to large formulae and consume more memory. The preconditions we generate do satisfy the interpolation condition and can be viewed as a restricted type of interpolants that have compact representations in addition to satisfying the usual vocabulary and implication conditions.

Our approach of generalising satisfying instances is in the same spirit as IC3. IC3 is a technique for incremental construction of inductive proofs for modular analysis [11]. Unlike IC3, BLITZ only seeks to discover assertion violations and does not attempt proofs of correctness.

There are two main strategies to apply scope-bounded analysis. One strategy partitions a program into multiple sub-programs at the outset (each with a subset of paths), and solve all of them in parallel [84]. Though amendable to a parallel architecture, it may generate a prohibitively large number of sub-programs. Another strategy is to overapproximate behavior outside the scope of bounded analysis, and iteratively refine it. Called 'structural abstraction' in CALYSTO [5], it also used in CORRAL [56] and ALTER [85]. DC2 further uses a lightweight static analysis to infer pre- and post-conditions to model the behavior of procedures outside the scope of analysis, but its scope expansion is not automatic. CORRAL automates forward (callee) scope expansion with stratified inlining of procedures and selective variable abstraction. ALTER alternates between eager forward

(callee) and lazy backward (caller) expansion, and uses interpolation to learn procedure preconditions that lead to failed scope expansions, to prevent re-exploration on backtracking. BLITZ recasts satisfying assignments to obtain unsatisfiable formulae, and then derives intermediate preconditions from unsatisfiability proofs. Unlike existing approaches where the size of a BMC instance grows as more procedures are *inlined*, BLITZ can bound the size of instances.

Slicing is commonly used to reduce the size of BMC instances by removing variables that are *syntactically* irrelevant to a property (e.g., [83, 73]). It is also known as 'cone of influence' reduction, where the set of relevant variables expands with the size of the slice, diminishing its effectiveness (e.g., despite slicing, CBMC and ESBMC frequently run out of time/memory). Beyond syntactic relevance, BLITZ also uses proofs of unsatisfiability to reason about *semantic* relevance on small code fragments at a time, producing an overall cone with much smaller base.

5.7 Conclusions and Future Work

We presented BLITZ, a new, compositional bounded model checking algorithm for software and showed that it scales BMC to real-world programs. BLITZ is capable of finding all known vulnerabilities in a known data set and has also discovered new vulnerabilities in widely deployed software. The new discoveries have led to bug fixes which will soon be deployed.

BLITZ works by constructing a series of BMC instances that when composed lead to a violation. A novel feature of our algorithm is the use of underapproximate preconditions in the context of BMC. The underapproximations are computed using information from resolution refutations generated by a SAT solver. The underapproximation guarantees that the tool does not generate false alarms about the existence of bugs and the proofs guarantee that the approximation is not too restricted by considering only facts relevant for the violation. Our procedure is parametric, allowing for the preconditions to be iteratively weakened to eventually derive Dijkstra's weakest precondition, if resources permit. The size of code fragments analysed is configurable and can range from a single statement to an entire procedure, depending on code-size and memory available.

There are several directions to explore. We did not use Craig interpolation over proofs because the formulae were large. An interesting question is whether proofs can be manipulated to yield interpolants that have compact representations and whether this improves performance. A second question is how one may guess the optimal configuration of the algorithm (unwinding, weakening, granularity) by a preliminary analysis of the code. Third, concrete and symbolic representations have been combined in symbolic execution with great success but such an approach has not been applied to BMC. We anticipate that combining concrete values with BMC may provide new opportunities for decomposition.

Chapter 6

Conclusions

In this thesis, we have developed techniques to automatically find bugs in network-based applications. We have explored two directions towards this goal: First, to overcome the problem of not knowing how an application interacts with its network environment, we have proposed new techniques to infer the protocol model implemented by an application, and use the inferred model to guide the search for bugs. We have shown that active automata learning algorithms such as L^* can be successfully applied to remotely infer the protocol model implemented by an application. We have applied our technique to infer the Command-and-Control (C&C) protocol model of a major spam botnet, and analyzed it to discover its weakest link, protocol logic flaws and unobservable communication between C&C servers. The critical link analysis has led to potentially more efficient ways to disable a botnet by targeting its weakest links, and the protocol logic flaws that we have discovered enable new ways to fight spam by collecting and using spam templates to filter spam, before any spam is generated.

Second, we have proposed new techniques to analyze programs to find bugs. Our techniques are based on a family of techniques called symbolic analysis (symbolic execution and bounded model checking), which is equivalent to testing with entire classes of test case inputs. We proposed MACE, which shows that a synergistic combination of model inference and symbolic execution is more effective than symbolic execution alone. MACE works by alternating between model inference and symbolic execution. MACE uses model inference to compute a model of program behavior, which guides state-space exploration through a mix of concrete and symbolic execution. During state-space exploration, MACE finds bugs and discovers new program behavior, which is used in turn to refine the inferred model. The two sub-components alternate till convergence. Using such a synergistic combination, MACE finds more vulnerabilities and explores more deeply into the application, discovering seven unique vulnerabilities in widely-used desktop applications, four of which are new.

One major limitation of symbolic analysis is lack of scalability — existing bounded model checking (BMC) tools run out of memory or time beyond programs with a few thousand lines of code. We proposed BLITZ, a new compositional bounded model checking technique that scales to large programs in the order of hundreds of thousands of lines of code (100 KLOC). BLITZ works by constructing a series of BMC instances that when composed lead to a violation. It uses a novel

combination of satisfying assignments and proofs of unsatisfiability to compute generalized underapproximations of preconditions that are guaranteed to lead to the violation. Using information from the preconditions for dataflow-based propagation, BLITZ analyzes only code fragments relevant for the violation. The preconditions can be weakened iteratively towards Dijkstra's weakest precondition, and the size of analyzed code fragments can be varied from a single statement to an entire procedure. BLITZ outperformed existing tools on a set of benchmarks with known vulnerabilities. Using the technique to analyze critical Internet infrastructure software used by Internet Service Providers, it discovered multiple new vulnerabilities.

We believe future research is needed in several directions. First, we have only just begun to exploit the synergies between learning a program's structure (model inference) and symbolic analysis, and there are richer ways to combine them that remains to be explored. For instance, BLITZ works by propagating preconditions backwards towards the program entry point, and might benefit from using the program's structure to guide propagation. Second, we have limited our model inference techniques to either the client or server side of a protocol. We believe it is fruitful to co-infer the protocol models at both sides in a modular fashion (e.g., in an assume-guarantee style). Third, we have not explored the application of our techniques for emerging network-based applications such as web and mobile, which will pose numerous new challenges and opportunities to be overcome.

Bibliography

[1] *2007 Malware Report: The Economic Impact of Viruses, Spyware, Adware, Botnets, and Other Malicious Code*. Tech. rep. Computer Economics Inc., 2007.

[2] Moheeb Abu Rajab, Jay Zarfoss, Fabian Monrose, and Andreas Terzis. "A multifaceted approach to understanding the botnet phenomenon". In: *IMC '06: Proc. of the 6th ACM SIGCOMM conference on Internet measurement*. Rio de Janeriro, Brazil: ACM, 2006, pp. 41–52.

[3] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. "Demand-driven compositional symbolic execution". In: *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 367–381. ISBN: 3-540-78799-2, 978-3-540-78799-0. URL: http://dl.acm.org/citation.cfm?id=1792734.1792771.

[4] Dana Angluin. "Learning regular sets from queries and counterexamples". In: *Information and Computation* 75.2 (1987), pp. 87–106.

[5] Domagoj Babic and Alan J. Hu. "Calysto: scalable and precise extended static checking". In: *Proceedings of the 30th international conference on Software engineering*. ICSE '08. Leipzig, Germany: ACM, 2008, pp. 211–220. ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368118. URL: http://doi.acm.org/10.1145/1368088.1368118.

[6] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram Rajamani. "Automatic Predicate Abstraction of C Programs". In: *PLDI'01: Proc. of the ACM SIGPLAN 2001 Conf. on Programming Language Design and Implementation*. Vol. 36. ACM SIGPLAN Notices. ACM Press, 2001, pp. 203–213.

[7] Mike Barnett, Robert Deline, Manuel Fähndrich, Bart Jacobs, K. Rustan Leino, Wolfram Schulte, and Herman Venter. "Verified Software: Theories, Tools, Experiments". In: Springer-Verlag, 2008. Chap. The Spec# Programming System: Challenges and Directions, pp. 144–152.

[8] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. 2009, pp. 825–885.

[9] Terry Benzel, Robert Braden, Dongho Kim, Clifford Neuman, Anthony Joseph, Keith Sklower, Ron Ostrenga, and Stephen Schwab. "Design, deployment, and use of the DETER testbed". In: *Proc. of the DETER Community Workshop on Cyber Security Experimentation and Test on DETER Community Workshop on Cyber Security Experimentation and Test*. USENIX Association, 2007.

[10] Nikita Borisov, David Brumley, Helen J. Wang, John Dunagan, Pallavi Joshi, and Chuanxiong Guo. "Generic Application-Level Protocol Analyzer and its Language". In: *NDSS'07: Proc. of the 2007 Network and Distributed System Security Symposium*. San Diego, CA, USA: The Internet Society, 2007.

[11] Aaron R. Bradley. "SAT-based model checking without unrolling". In: *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation*. VMCAI'11. Austin, TX, USA: Springer-Verlag, 2011, pp. 70–87. ISBN: 978-3-642-18274-7. URL: http://dl.acm.org/citation.cfm?id=1946284.1946291.

[12] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. "An Infrastructure for Adaptive Dynamic Optimization". In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '03. San Francisco, California: IEEE Computer Society, 2003, pp. 265–275. ISBN: 0-7695-1913-X. URL: http://dl.acm.org/citation.cfm?id=776261.776290.

[13] Juan Caballero, Noah M. Johnson, Stephen McCamant, and Dawn Song. "Binary Code Extraction and Interface Identification for Security Applications". In: *NDSS'10: Proceedings of the 17th Annual Network and Distributed System Security Symposium*. San Diego, CA, USA, 2010.

[14] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. "Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering". In: *CCS'09: Proc. of the 16th ACM conference on Computer and communications security*. Chicago, IL, USA: ACM, 2009, pp. 621–634.

[15] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis". In: *CCS'07: Proc. of the 14th ACM Conf. on Computer and Communications Security*. Alexandria, VI, USA: ACM, 2007, pp. 317–329.

[16] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *OSDI'08: Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation*. San Diego, California, USA: USENIX Association, 2008, pp. 209–224.

[17] Cristian Cadar and Dawson R. Engler. "Execution Generated Test Cases: How to Make Systems Code Crash Itself". In: *SPIN'05: Proc. of the 12th Int. SPIN Workshop on Model Checking Software*. Vol. 3639. Lecture Notes in Computer Science. San Francisco, California, USA: Springer, 2005, pp. 2–23.

[18] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. "EXE: Automatically Generating Inputs of Death". In: *ACM Trans. Inf. Syst. Secur.* 12 (2 2008), pp. 1–38.

[19] Chia Yuan Cho, Domagoj Babić, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. "MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery". In: *Proceedings of the 20th USENIX Security Symposium*. San Francisco, CA, Aug. 2011.

[20] Chia Yuan Cho, Domagoj Babić, Richard Shin, and Dawn Song. "Inference and Analysis of Formal Models of Botnet Command and Control Protocols". In: *CCS'10: Proc. of the 2010 ACM Conf. on Computer and Communications Security*. Chicago, Illinois, USA: ACM, 2010, pp. 426–440.

[21] Chia Yuan Cho, Juan Caballero, Chris Grier, Vern Paxson, and Dawn Song. "Insights from the inside: a view of botnet management from infiltration". In: *Proceedings of the 3rd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*. LEET'10. San Jose, California: USENIX Association, 2010. URL: http://dl.acm.org/citation.cfm?id=1855686.1855688.

[22] Chia Yuan Cho, Vijay D'Silva, and Dawn Song. "Blitz: Compositional Bounded Model Checking for Real-world Programs". In: *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. Palo Alto, CA, Nov. 2013.

[23] Edmund Clarke, Daniel Kroening, and Karen Yorav. "Behavioral consistency of C and verilog programs using bounded model checking". In: *Proceedings of the 40th annual Design Automation Conference*. DAC '03. Anaheim, CA, USA: ACM, 2003, pp. 368–371. ISBN: 1-58113-688-9. DOI: 10.1145/775832.775928. URL: http://doi.acm.org/10.1145/775832.775928.

[24] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-Guided Abstraction Refinement". In: *Proceedings of the 12th International Conference on Computer Aided Verification*. CAV '00. London, UK, UK: Springer-Verlag, 2000, pp. 154–169. ISBN: 3-540-67770-4. URL: http://dl.acm.org/citation.cfm?id=647769.734089.

[25] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 1999.

[26] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. "Learning assumptions for compositional verification". In: *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*. TACAS'03. Warsaw, Poland: Springer-Verlag, 2003, pp. 331–346. ISBN: 3-540-00898-5. URL: http://dl.acm.org/citation.cfm?id=1765871.1765903.

[27] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. "Prospex: Protocol Specification Extraction". In: *S&P'09: Proc. of the 2009 30th IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2009, pp. 110–125.

[28] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 2nd. The MIT Press, 2001.

[29] Weidong Cui, Jayanthkumar Kannan, and Helen J. Wang. "Discoverer: Automatic protocol reverse engineering from network traces". In: *Proc. of 16th USENIX Security Symposium*. Boston, MA, USA: USENIX Association, 2007, pp. 1–14.

[30] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irún-Briz. "Tupni: Automatic reverse engineering of input formats". In: *CCS'08: Proc. of the 15th ACM Conf. on Computer and Communications Security*. Alexandria, VI, USA: ACM, 2008, pp. 391–402.

[31] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "An Efficient Method of Computing Static Single Assignment Form". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: ACM, 1989, pp. 25–35. ISBN: 0-89791-294-2. DOI: 10.1145/75277.75280. URL: http://doi.acm.org/10.1145/75277.75280.

[32] Roger Dingledine, Nick Mathewson, and Paul Syverson. "Tor: the second-generation onion router". In: *SSYM'04: Proc. of the 13th conference on USENIX Security Symposium*. San Diego, CA: USENIX Association, 2004, pp. 21–21.

[33] Niklas Eén and Niklas Sörensson. "An Extensible SAT-solver". In: *SAT*. Ed. by Enrico Giunchiglia and Armando Tacchella. Vol. 2919. Lecture Notes in Computer Science. Springer, 2003, pp. 502–518. ISBN: 3-540-20851-8.

[34] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing". In: *PLDI'05: Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*. Chicago, Illinois, USA: ACM, 2005, pp. 213–223.

[35] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. "Automated Whitebox Fuzz Testing". In: *NDSS'08: Proc. of the Network and Distributed System Security Symposium*. San Diego, California, USA: The Internet Society, 2008.

[36] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. "Compositional may-must program analysis: unleashing the power of alternation". In: *SIGPLAN Not.* 45.1 (Jan. 2010), pp. 43–56. ISSN: 0362-1340. DOI: 10.1145/1707801.1706307. URL: http://doi.acm.org/10.1145/1707801.1706307.

[37] E. Mark Gold. "Complexity of automaton identification from given data". In: *Information and Control* 37.3 (1978), pp. 302–320.

[38] Julian B. Grizzard, Vikram Sharma, Chris Nunnery, Brent ByungHoon Kang, and David Dagon. "Peer-to-peer botnets: overview and case study". In: *HotBots'07: Proc. of the 1st Workshop on Hot Topics in Understanding Botnets*. Cambridge, MA, USA: USENIX Association, 2007, pp. 1–1.

[39] Guofei Gu, Roberto Perdisci, Junjie Zhang, and Wenke Lee. "BotMiner: clustering analysis of network traffic for protocol- and structure-independent botnet detection". In: *Proc. of the 17th conference on Security symposium*. San Jose, CA, USA: USENIX Association, 2008, pp. 139–154.

[40] Guofei Gu, Phillip Porras, Vinod Yegneswaran, Martin Fong, and Wenke Lee. "BotHunter: detecting malware infection through IDS-driven dialog correlation". In: *Proc. of 16th USENIX Security Symposium on USENIX Security Symposium*. Boston, MA, USA: USENIX Association, 2007, pp. 1–16.

[41] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. "SYNERGY: a new algorithm for property checking". In: *FSE'06: Proc. of the 14th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*. ACM, 2006, pp. 117–127.

[42] Anubhav Gupta. "Learning Abstractions for Model Checking". PhD thesis. School of Computer Science, Carnegie Mellon University, 2006. URL: `http://reports-archive.adm.cs.cmu.edu/anon/2006/CMU-CS-06-131.pdf`.

[43] Anubhav Gupta, K. L. McMillan, and Zhaohui Fu. "Automated assumption generation for compositional verification". In: *Form. Methods Syst. Des.* 32.3 (2008), pp. 285–301.

[44] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. "Software Verification with Blast". In: *SPIN'03: Proc. of the 10th Int. Workshop on Model Checking of Software*. Vol. 2648. LNCS. Springer-Verlag, 2003, pp. 235–239.

[45] Colin de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.

[46] Pei Hsin Ho, Thomas Shiple, Kevin Harer, James Kukula, Robert Damiano, Valeria Bertacco, Jerry Taylor, and Jiang Long. "Smart simulation using collaborative formal and simulation engines". In: *ICCAD'00: Proc. of the 2000 IEEE/ACM Int. Conf. on Computer-aided design*. IEEE Press, 2000, pp. 120–126.

[47] C. A. R. Hoare. "An axiomatic basis for computer programming". In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: `10.1145/363235.363259`. URL: `http://doi.acm.org/10.1145/363235.363259`.

[48] Tating Hsu, Guoqiang Shu, and David Lee. "A Model-based Approach to Security Flaw Detection of Network Protocol Implementation". In: *ICNP'08: Proc. of the 15th IEEE Int. Conf. on Network Protocols*. Orlando, FL, USA, 2008, pp. 114–123.

[49] Rauli Kaksonen. *A Functional Method for Assessing Protocol Implementation Security*. VTT Publications 448, 2001. ISBN: 951-38-5874-X.

[50] Anestis Karasaridis, Brian Rexroad, and David Hoeflin. "Wide-scale botnet detection and characterization". In: *HotBots'07: Proc. of the 1st Workshop on Hot Topics in Understanding Botnets*. Cambridge, MA, USA: USENIX Association, 2007, pp. 7–7.

[51] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. "Generalized symbolic execution for model checking and testing". In: *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*. TACAS'03. Warsaw, Poland: Springer-Verlag, 2003, pp. 553–568. ISBN: 3-540-00898-5. URL: `http://dl.acm.org/citation.cfm?id=1765871.1765924`.

[52] James C. King. “Symbolic execution and program testing”. In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[53] J. Klensin. *RFC 5321: Simple Mail Transfer Protocol*. 2008.

[54] Daniel Kroening and Georg Weissenbacher. “Interpolation-based software verification with WOLVERINE”. In: *Proceedings of the 23rd international conference on Computer aided verification*. CAV’11. Snowbird, UT: Springer-Verlag, 2011, pp. 573–578. ISBN: 978-3-642-22109-5. URL: http://dl.acm.org/citation.cfm?id=2032305.2032350.

[55] Andreas Kuehlmann and Florian Krohm. “Equivalence checking using cuts and heaps”. In: *DAC’97: Proc. of the 34th annual Design Automation Conference*. Anaheim, California, United States: ACM, 1997, pp. 263–268.

[56] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. “A solver for reachability modulo theories”. In: *Proceedings of the 24th international conference on Computer Aided Verification*. CAV’12. Berkeley, CA: Springer-Verlag, 2012, pp. 427–443. ISBN: 978-3-642-31423-0. DOI: 10.1007/978-3-642-31424-7_32. URL: http://dx.doi.org/10.1007/978-3-642-31424-7_32.

[57] Kevin J. Lang. *Faster Algorithms for Finding Minimal Consistent DFAs*. Tech. rep. NEC, 1999.

[58] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO ’04. Palo Alto, California: IEEE Computer Society, 2004, pp. 75–. ISBN: 0-7695-2102-9. URL: http://dl.acm.org/citation.cfm?id=977395.977673.

[59] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. “Bugbench: Benchmarks for evaluating bug detection tools”. In: *In Workshop on the Evaluation of Software Defect Detection Tools*. 2005.

[60] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation”. In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’05. Chicago, IL, USA: ACM, 2005, pp. 190–200. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065034. URL: http://doi.acm.org/10.1145/1065010.1065034.

[61] Inês Lynce and João P. Marques Silva. “On Computing Minimum Unsatisfiable Cores”. In: *Conference on Theory and Applications of Satisfiability Testing*. Online Proceedings, 2004.

[62] Sharad Malik and Lintao Zhang. “Boolean Satisfiability from Theoretical Hardness to Practical Success”. In: *Commun. ACM* 52.8 (Aug. 2009), pp. 76–82. ISSN: 0001-0782. DOI: 10.1145/1536616.1536637. URL: http://doi.acm.org/10.1145/1536616.1536637.

[63] K. L. McMillan. “Relevance heuristics for program analysis”. In: *Symposium on Principles of programming languages*. ACM, 2008, pp. 145–146.

[64] Kenneth L. McMillan. “Interpolation and SAT-Based Model Checking”. In: *CAV’03*. 2003, pp. 1–13.

[65] Kenneth L. McMillan. “Lazy abstraction with interpolants”. In: *Proceedings of the 18th international conference on Computer Aided Verification*. CAV’06. Seattle, WA: Springer-Verlag, 2006, pp. 123–136. ISBN: 3-540-37406-X, 978-3-540-37406-0. DOI: 10.1007/11817963_14. URL: http://dx.doi.org/10.1007/11817963_14.

[66] Kenneth L. McMillan. “Lazy Annotation for Program Testing and Verification”. In: *Computer Aided Verification*. 2010, pp. 104–118.

[67] George H. Mealy. “A Method for Synthesizing Sequential Circuits”. In: *Bell System Technical Journal* 34.5 (1955), pp. 1045–1079.

[68] Mehryar Mohri and Mark-Jan Nederhof. “Regular Approximation of Context-Free Grammars through Transformation”. In: *Robustness in Language and Speech Technology*. Kluwer Academic Publishers, 2001, pp. 153–163.

[69] E. F. Moore. “Gedanken Experiments On Sequential Machines”. In: *Automata Studies, Annals of Mathematical Studies*. Vol. 34. Princeton University Press, 1956, pp. 129–153.

[70] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *TACAS’08: Proc. of the 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340.

[71] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. “CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs”. In: *Proceedings of the 11th International Conference on Compiler Construction*. CC ’02. London, UK, UK: Springer-Verlag, 2002, pp. 213–228. ISBN: 3-540-43369-4. URL: http://dl.acm.org/citation.cfm?id=647478.727796.

[72] Nicholas Nethercote and Julian Seward. “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation”. In: *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’07. San Diego, California, USA: ACM, 2007, pp. 89–100. ISBN: 978-1-59593-633-2. DOI: 10.1145/1250734.1250746. URL: http://doi.acm.org/10.1145/1250734.1250746.

[73] Bruno Cuervo Parrino, Juan Pablo Galeotti, Diego Garbervetsky, and Marcelo F. Frias. “A dataflow analysis to improve SAT-based bounded program verification”. In: *Proceedings of the 9th international conference on Software engineering and formal methods*. SEFM’11. Montevideo, Uruguay: Springer-Verlag, 2011, pp. 138–154. ISBN: 978-3-642-24689-0. URL: http://dl.acm.org/citation.cfm?id=2075679.2075692.

[74] Vern Paxson. “Bro: a system for detecting network intruders in real-time”. In: *SSYM’98: Proc. of the 7th conference on USENIX Security Symposium*. San Antonio, TX, USA: USENIX Association, 1998, pp. 3–3.

[75] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. "Black Box Checking". In: *Proc. of the IFIP TC6 WG6.1 Joint Int. Conf. on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX)*. Kluwer, B.V., 1999, pp. 225–240.

[76] C. P. Pfleeger. "State Reduction in Incompletely Specified Finite-State Machines". In: *IEEE Transactions on Computers* 22.12 (1973), pp. 1099–1102.

[77] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software". In: *Proceedings of the 2008 international symposium on Software testing and analysis*. ISSTA '08. Seattle, WA, USA: ACM, 2008, pp. 15–26. ISBN: 978-1-60558-050-0. DOI: 10.1145/1390630.1390635. URL: http://doi.acm.org/10.1145/1390630.1390635.

[78] J. R. Quinlan. "Induction of Decision Trees". In: *Machine Learning* 1.1 (1986), pp. 81–106.

[79] R. L. Rivest and R. E. Schapire. "Inference of finite automata using homing sequences". In: *STOC'89: Proc. of the 21st annual ACM symposium on Theory of computing*. Seattle, Washington, United States: ACM, 1989, pp. 411–420.

[80] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: a concolic unit testing engine for C". In: *SIGSOFT Softw. Eng. Notes* 30 (5 2005), pp. 263–272.

[81] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. "AddressSanitizer: A Fast Address Sanity Checker". In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC'12. Boston, MA: USENIX Association, 2012, pp. 28–28. URL: http://dl.acm.org/citation.cfm?id=2342821.2342849.

[82] Muzammil Shahbaz and Roland Groz. "Inferring Mealy Machines". In: *FM'09: Proc. of the 2nd World Congress on Formal Methods*. Eindhoven, The Netherlands: Springer, 2009, pp. 207–222.

[83] Danhua Shao, Divya Gopinath, Sarfraz Khurshid, and Dewayne E. Perry. "Optimizing Incremental Scope-Bounded Checking with Data-Flow Analysis". In: *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*. ISSRE '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 408–417. ISBN: 978-0-7695-4255-3. DOI: 10.1109/ISSRE.2010.27. URL: http://dx.doi.org/10.1109/ISSRE.2010.27.

[84] Danhua Shao, Sarfraz Khurshid, and Dewayne E. Perry. "An Incremental Approach to Scope-Bounded Checking Using a Lightweight Formal Method". In: *Proceedings of the 2nd World Congress on Formal Methods*. FM '09. Eindhoven, The Netherlands: Springer-Verlag, 2009, pp. 757–772. ISBN: 978-3-642-05088-6. DOI: 10.1007/978-3-642-05089-3_48. URL: http://dx.doi.org/10.1007/978-3-642-05089-3_48.

[85] Nishant Sinha, Nimit Singhania, Satish Chandra, and Manu Sridharan. "Alternate and learn: finding witnesses without looking all over". In: *Proceedings of the 24th international conference on Computer Aided Verification*. CAV'12. Berkeley, CA: Springer-Verlag, 2012, pp. 599–615. ISBN: 978-3-642-31423-0. DOI: 10.1007/978-3-642-31424-7_42. URL: http://dx.doi.org/10.1007/978-3-642-31424-7_42.

[86] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. "Secure Program Execution via Dynamic Information Flow Tracking". In: *SIGARCH Comput. Archit. News* 32.5 (Oct. 2004), pp. 85–96. ISSN: 0163-5964. DOI: 10.1145/1037947.1024404. URL: http://doi.acm.org/10.1145/1037947.1024404.

[87] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternel war in memory". In: *Proc. Int'l Symp. on Security and Privacy, 2013*. San Francisco, USA, May 2013.

[88] B. A. Trakhtenbrot and Ya. M. Barzdin. *Finite Automata, Behavior and Synthesis*. North Holland, 1973.

[89] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. "Formal methods and testing". In: Springer-Verlag, 2008. Chap. Model-based testing of object-oriented reactive systems with spec explorer, pp. 39–76.

[90] Juan Miguel Vilar. "Query learning of subsequential transducers". In: *Proc. of the 3rd Int. Colloquium on Grammatical Inference: Learning Syntax from Sentences*. Springer-Verlag, 1996, pp. 72–83.

[91] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. "Shield: vulnerability-driven network filters for preventing known vulnerability exploits". In: *SIGCOMM Computer Communication Review* 34.4 (2004), pp. 193–204.

[92] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. "ReFormat: Automatic Reverse Engineering of Encrypted Messages". In: *ESORICS'09: 14th European Symposium on Research in Computer Security*. Vol. 5789. Lecture Notes in Computer Science. Saint-Malo, France: Springer, 2009, pp. 200–215.

[93] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. "Efficient conflict driven learning in a boolean satisfiability solver". In: *ICCAD'01: Proc. of the Int. Conf. on Computer-Aided Design*. IEEE Press, 2001, pp. 279–285.