MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery

Chia Yuan Cho[†‡] Domagoj Babić[†] Pongsin Poosankam[†§]

Kevin Zhijie Chen[†] Edward XueJun Wu[†] Dawn Song[†]

[†]*University of California, Berkeley* [§]*Carnegie Mellon University* [‡]*DSO National Labs*

Abstract

Program state-space exploration is central to software security, testing, and verification. In this paper, we propose a novel technique for state-space exploration of software that maintains an ongoing interaction with its environment. Our technique uses a combination of symbolic and concrete execution to build an abstract model of the analyzed application, in the form of a finite-state automaton, and uses the model to guide further state-space exploration. Through exploration, MACE further refines the abstract model. Using the abstract model as a scaffold, our technique wields more control over the search process. In particular: (1) shifting search to different parts of the search-space becomes easier, resulting in higher code coverage, and (2) the search is less likely to get stuck in small local state-subspaces (e.g., loops) irrelevant to the application's interaction with the environment. Preliminary experimental results show significant increases in the code coverage and exploration depth. Further, our approach found a number of new deep vulnerabilities.

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 paper, 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 [22] to explore the program's state-space, and (2) the L^* [1] 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 [17, 25, 8, 7], 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., [30]), dynamic symbolic execution only tracks the most promising path prefix for the next iteration [17], but does not learn in the sense that informa-

[§]This work was done while Pongsin Poosankam was a visiting student at UC Berkeley.

tion 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.

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# [3], for describing abstract software models. Such models can be used effectively as constraints for generating tests [27], but have to be written manually, which is both time consuming and requires a high level of expertise.

Grammar inference (e.g., [16]) promises automatic inference of models, and has been an active area of research in security, especially applied to protocol inference. Comparetti et al. [12] 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 [18, 25, 9, 7] 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 [10], 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 nontrivial and requires multiple iterations, especially for proprietary protocols, for which specifications are not available. In this paper, 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 is a purely black-box approach, while in this paper we do code analysis at the binary level in combination with grammatical inference.

In this paper, 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. [14, 15] and Caballero et al. [6] proposed approaches that could be used for that purpose. Further, any automatic protocol inference technique has to deal with encryption. In this paper, 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. [5] and Wang et al. [29] addresses automatic reverse-engineering of encrypted messages.

Software model checking tools, like SLAM [2] and Blast [20], incrementally build predicate abstractions of the analyzed software, but such abstractions are very different from the models inferred by the protocol inference techniques [12, 11]. 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 [19] 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 [21] 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^* [1] 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.

3 Problem Definition and Overview

We begin this section with the problem statement and a list of assumptions that we make in this paper. 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 paper.

3.1 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.

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,

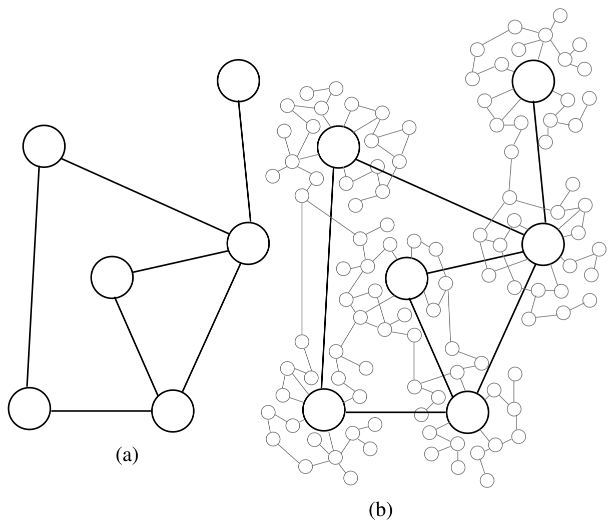


Figure 1: An Abstract Rendition of the MACE State-Space Exploration. The figure on the left shows an abstract model, i.e., a finite-state machine, inferred by MACE. The figure on the right depicts clusters of concrete states of the analyzed application, such that clusters are abstracted with a single 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.

which can result in inference of new transitions and states. Black box inference algorithms, like L^* [1], 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 paper 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 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.

3.2 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.

3.3 Preliminaries

Following our prior work [10], we use *Mealy machines* [23] 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:

Definition 1 (Mealy Machine)**.** *A Mealy machine, M, is a six-tuple* $(Q, \Sigma_I, \Sigma_O, \delta, \lambda, q_0)$*, 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),* $\delta : Q \times \Sigma_I \longrightarrow Q$ *is the transition relation, and* $\lambda : Q \times \Sigma_I \longrightarrow \Sigma_O$ *is the output relation.*

We extend the δ and λ relations to sequences of messages $m_j \in \Sigma_I$ as usual, e.g., $\delta(q, m_0 \cdot m_1 \cdot m_2) = \delta(\delta(\delta(q, m_0), m_1), m_2)$ and $\lambda(q, m_0 \cdot m_1 \cdot m_2) = \lambda(q, m_0) \cdot \lambda(\delta(q, m_0), m_1) \cdot \lambda(\delta(q, m_0 \cdot m_1), m_2)$. 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 $sup(s) = \{s_j \mid 0 \le j < |s|\}$. If for some state machine $M = (Q, \Sigma_I, \Sigma_O, \delta, \lambda, q_0)$ and some state $q \in Q$ there is $s \in \Sigma_I^*$ such that $\delta(q_0, s) = q$, 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:

Definition 2 (Redundant Input Symbols)**.** *Let* $M = (Q, \Sigma_I, \Sigma_O, \delta, \lambda, q_0)$ *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* $\forall q \in Q \,.\, \lambda(q, m) = \lambda(q, m') \wedge \delta(q, m) = \delta(q, m')$.

We say that a Mealy machine $M = (Q, \Sigma_I, \Sigma_O, \delta, \lambda, q_0)$ is complete iff $\delta(q, i)$ and $\lambda(q, i)$ are defined for every $q \in Q$ and $i \in \Sigma_I$. In this paper, 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 [26] variant of the classical L^* [1] 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 [1] on the accuracy of the inferred state machine. The third approach, called black box model checking [24], uses bounded model checking to compare the conjecture with the black box.

As discussed in Section 3.1, 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 [10], 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 $\alpha_O(o) = \alpha_O(o_0) \cdots \alpha_O(o_{n-1})$.

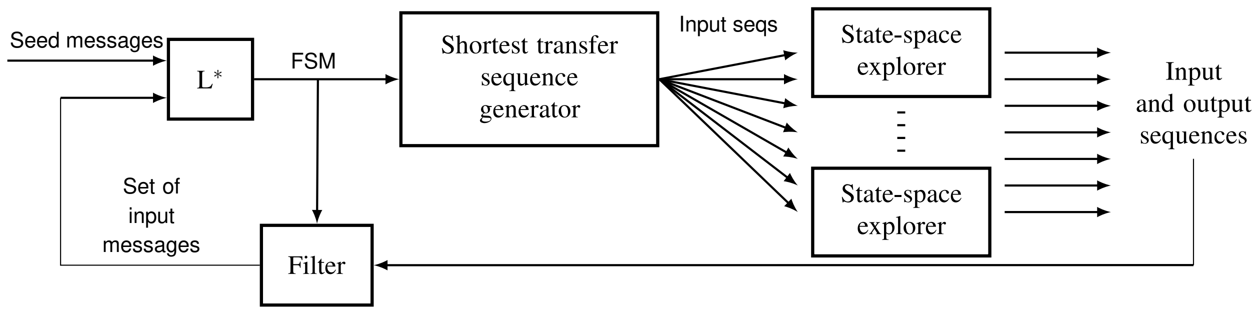


Figure 2: The MACE Approach Diagram. The L^* algorithm takes in the input and output alphabets, over which it infers a state-machine. L^* sends queries and receives responses from the analyzed application, which is not shown in the figure. The result of inference is a finite-state machine (FSM). For every state in the inferred state machine, We generate a shortest transfer sequence (Section 3.3) 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.

4 Model-inference-Assisted Concolic Exploration

We begin this section by a high-level description of MACE, illustrated in Figure 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.

4.1 A High-Level Description

Suppose we want to infer a complete Mealy machine $M = (Q, \Sigma_I, \Sigma_O, \delta, \lambda, q_0)$ 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 [17, 25, 8, 7]. The initial Σ_{I0} alphabet could be empty, but MACE would take longer to converge. In our work, we used regression test suites provided 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 $M_0 = (Q_0, \Sigma_{I0}, \Sigma_O, \delta_0, \lambda_0, q_0^0)$ 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. 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.

4.2 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 [26] variant of L^* for learning Mealy machines. The inference process is similar as in our prior work [10].

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 3.3, 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.

4.3 The State-Space Exploration Phase

We use the model inferred in Section 4.2 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^i_0. 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 $q^i_0 \xrightarrow{+} q^i$, 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 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.

4.4 Model Refinement

The exploration phase described in Section 4.3 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. [12] propose a message clustering technique, while we used a handcrafted an abstraction function in our prior work. In this paper, 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:

Definition 3 (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:*

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

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 [28], 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 $\Sigma_{I(i+1)}$.

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 5.1 explains this interaction in more detail. Section 5.2 surveys the main model inference optimizations, including parallelization, caching, and filtering. Finally, Section 5.3 introduces our state-space exploration component, which is used as a baseline for the later provided experimental results.

5.1 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 nondeterministic 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.

5.2 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 2) 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.

5.3 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 [1]) 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 [17, 25, 8, 7]. 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 [18], 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.

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. 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 6.2.

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 6.3. In Section 6.4, we evaluate the effectiveness of MACE, by comparing it to the baseline state-space exploration component of MACE without guidance.

6.1 Experimental Setup

For our state-space exploration experiments, we used the DETER Security testbed [4] 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.

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

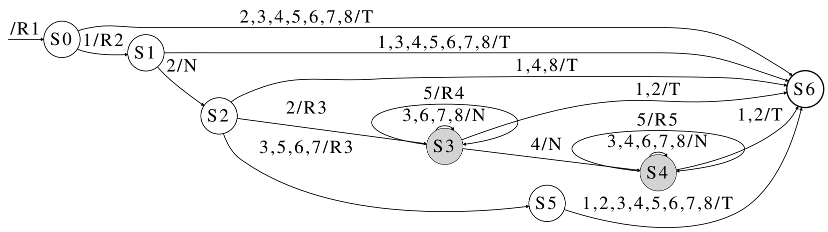
Table 1: Model Inference Result at the End of Each Iteration. The second column identifies the inference iteration. The Q column denotes the number of states in the inferred model. The Σ_I (resp. Σ_O) column denotes the size of the input (resp. output) alphabet. The last column gives the total time (sum 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.

6.2 Model Inference and Refinement

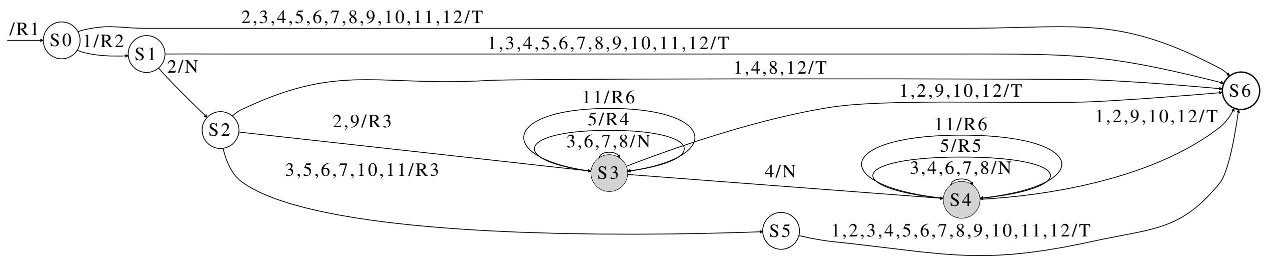
We used MACE to iteratively infer and refine the protocol models of RFB and SMB, using Vino 2.26.1 and Samba 3.3.4 as reference implementations respectively. Table 1 shows the results of iterative model inference and refinement on Vino and Samba.

As discussed in Section 4.2, 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



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



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

Figure 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.

messages, as shown in Figure 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 3b). We manually inspected the newly discovered output message (label R6 in Figure 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.

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.

Samba. For Samba, we collected a network trace of multiple SMB sessions, using Samba's `gentest` test suite, 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).

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 1 summarizes all three inference rounds.

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

There are two other output message types that are triggered by the server's GUI events and thus are outside of our scope.

http://samba.org/∼tridge/samba_testing/

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: Explanation of States and Input/Output Messages of the State Machine from Figure 3.

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 3.1).

6.3 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 3.3) 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, 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,

http://valgrind.org/

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 2. 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 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, 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 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 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.

6.4 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

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

Program	Vulnerability Type	Disclosure ID	Iter.	Jobs ($\lvert Q \rvert$)	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
Win XP SMB	None	None	None	84	>150	>210	>900	>105

Table 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.

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 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 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.

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 6.1. 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 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 2) 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.

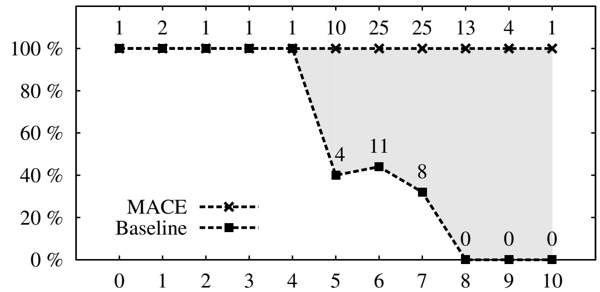


Figure 5: SMB Exploration Depth. The inferred state machine can be seen as a directed graph. Suppose we compute a spanning tree (e.g., [13]) of that graph. The root of the graph is at level zero. Its children are at level one, and so on. The figure shows the percentage of states visited at each level 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.

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., [13]) 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 5 clearly shows that MACE is superior to the baseline approach in reaching deep states in the inferred protocol.

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 2).

Our approach to model inference and refinement is not entirely automatic: the end users need 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 [11], 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 [14, 15, 6] out of the scope of this paper.

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 [5, 29] that automatically reverse-engineer message encryption are required.

8 Acknowledgements

We would like to thank the anonymous reviewers for their insightful comments to improve this manuscript. This material is based upon work partially supported by the NSF under Grants No. 0311808, 0832943, 0448452, 0842694, 0627511, 0842695, 0831501, and 0424422, by the AFRL under Grant No. P010071555, by the ONR under MURI Grant No. N000140911081, and by the MURI program under AFOSR Grants No. FA9550-08-1-0352 and FA9550-09-1-0539. The second author is also supported by the NSERC (Canada) PDF fellowship.

9 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 paper. 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 paper 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 [28] 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.

References

[1] ANGLUIN, D. Learning regular sets from queries and counterexamples. *Information and Computation 75*, 2 (1987), 87–106.

[2] BALL, T., MAJUMDAR, R., MILLSTEIN, T., AND RAJAMANI, S. Automatic Predicate Abstraction of C Programs. In *PLDI'01: Proc. of the ACM SIGPLAN 2001 Conf. on Programming Language Design and Implementation* (2001), vol. 36 of *ACM SIGPLAN Notices*, ACM Press, pp. 203–213.

[3] BARNETT, M., DELINE, R., FÄHNDRICH, M., JACOBS, B., LEINO, K. R., SCHULTE, W., AND VENTER, H. Verified software: Theories, tools, experiments. Springer-Verlag, 2008, ch. The Spec# Programming System: Challenges and Directions, pp. 144–152.

[4] BENZEL, T., BRADEN, R., KIM, D., NEUMAN, C., JOSEPH, A., SKLOWER, K., OSTRENGA, R., AND SCHWAB, S. 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* (2007), USENIX Association.

[5] CABALLERO, J., POOSANKAM, P., KREIBICH, C., AND SONG, D. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In

CCS'09: Proc. of the 16th ACM conference on Computer and communications security (2009), ACM, pp. 621–634.

[6] CABALLERO, J., YIN, H., LIANG, Z., AND SONG, D. 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* (2007), ACM, pp. 317–329.

[7] CADAR, C., DUNBAR, D., AND ENGLER, D. 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* (2008), USENIX Association, pp. 209–224.

[8] CADAR, C., AND ENGLER, D. R. 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* (2005), vol. 3639 of *Lecture Notes in Computer Science*, Springer, pp. 2–23.

[9] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur. 12* (2008), 1–38.

[10] CHO, C. Y., BABIĆ, D., SHIN, R., AND SONG, D. 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* (2010), ACM, pp. 426–440.

[11] CHO, C. Y., CABALLERO, J., GRIER, C., PAXSON, V., AND SONG, D. Insights from the inside: A view of botnet management from infiltration. In *LEET'10: Proc. of the 3rd USENIX Workshop on Large-Scale Exploits and Emergent Threats* (2010), USENIX Association, pp. 1–1.

[12] COMPARETTI, P. M., WONDRACEK, G., KRUEGEL, C., AND KIRDA, E. Prospex: Protocol specification extraction. In *S&P'09: Proc. of the 2009 30th IEEE Symposium on Security and Privacy* (2009), IEEE Computer Society, pp. 110–125.

[13] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms*, 2nd ed. The MIT Press, 2001.

[14] CUI, W., KANNAN, J., AND WANG, H. J. Discoverer: Automatic protocol reverse engineering from network traces. In *Proc. of 16th USENIX Security Symposium* (2007), USENIX Association, pp. 1–14.

[15] CUI, W., PEINADO, M., CHEN, K., WANG, H. J., AND IRÚN-BRIZ, L. Tupni: Automatic reverse engineering of input formats. In *CCS'08: Proc. of the 15th ACM Conf. on Computer and Communications Security* (2008), ACM, pp. 391–402.

[16] DE LA HIGUERA, C. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.

[17] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: directed automated random testing. In *PLDI'05: Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation* (2005), ACM, pp. 213–223.

[18] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. A. Automated whitebox fuzz testing. In *NDSS'08: Proc. of the Network and Distributed System Security Symposium* (2008), The Internet Society.

[19] GULAVANI, B. S., HENZINGER, T. A., KANNAN, Y., NORI, A. V., AND RAJAMANI, S. K. SYNERGY: a new algorithm for property checking. In *FSE'06: Proc. of the 14th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering* (2006), ACM, pp. 117–127.

[20] HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND SUTRE, G. Software Verification with Blast. In *SPIN'03: Proc. of the 10th Int. Workshop on Model Checking of Software* (2003), vol. 2648 of *LNCS*, Springer-Verlag, pp. 235–239.

[21] HO, P. H., SHIPLE, T., HARER, K., KUKULA, J., DAMIANO, R., BERTACCO, V., TAYLOR, J., AND LONG, J. Smart simulation using collaborative formal and simulation engines. In *ICCAD'00: Proc. of the 2000 IEEE/ACM Int. Conf. on Computer-aided design* (2000), IEEE Press, pp. 120–126.

[22] KING, J. C. Symbolic execution and program testing. *Communications of the ACM 19*, 7 (1976), 385–394.

[23] MEALY, G. H. A method for synthesizing sequential circuits. *Bell System Technical Journal 34*, 5 (1955), 1045–1079.

[24] PELED, D., VARDI, M. Y., AND YANNAKAKIS, M. 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)* (1999), Kluwer, B.V., pp. 225–240.

[25] SEN, K., MARINOV, D., AND AGHA, G. Cute: a concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes 30* (2005), 263–272.

[26] SHAHBAZ, M., AND GROZ, R. Inferring Mealy machines. In *FM'09: Proc. of the 2nd World Congress on Formal Methods* (2009), Springer, pp. 207–222.

[27] VEANES, M., CAMPBELL, C., GRIESKAMP, W., SCHULTE, W., TILLMANN, N., AND NACHMANSON, L. Formal methods and testing. Springer-Verlag, 2008, ch. Model-based testing of object-oriented reactive systems with spec explorer, pp. 39–76.

[28] VILAR, J. M. Query learning of subsequential transducers. In *Proc. of the 3rd Int. Colloquium on Grammatical Inference: Learning Syntax from Sentences* (1996), Springer-Verlag, pp. 72–83.

[29] WANG, Z., JIANG, X., CUI, W., WANG, X., AND GRACE, M. ReFormat: Automatic reverse engineering of encrypted messages. In *ESORICS'09: 14th European Symposium on Research in Computer Security* (2009), vol. 5789 of *Lecture Notes in Computer Science*, Springer, pp. 200–215.

[30] ZHANG, L., MADIGAN, C. F., MOSKEWICZ, M. H., AND MALIK, S. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD'01: Proc. of the Int. Conf. on Computer-Aided Design* (2001), IEEE Press, pp. 279–285.