Verification State-Space Reduction Through Restricted Parsing Environments

Jacob I. Torrey and Mark P. Bridgman
Assured Information Security
Greenwood Village, CO, USA
{*torreyj, bridgmanm*}*@ainfosec.com*

***Abstract*—We discuss the potential for significant reduction in size and complexity of verification tasks for input-handling software when such software is constructed according to LangSec principles, i.e., is designed as a recognizer for a particular language of valid inputs and is compiled for a suitably limited computational model no stronger than needed for the recognition task. We introduce *Crema*, a programming language and restricted execution environment of sub-Turing power, and conduct a case study to estimate and compare the respective sizes of verification tasks for the *qmail* SMTP parsing code fragments when executed natively vs in Crema—using LLVM and KLEE. We also study the application of the same principles to the verification of reference monitors.**

***Keywords*-Walther recursion, verification, qmail, programming languages, parsing, LLVM, language-theoretic security, LangSec.**

I. Introduction

Language-theoretic security (LangSec) posits that input-handling code modules must be constructed as verifiable *recognizers* for well-specified languages of valid or expected inputs, described by means of the formal language theory [1]. In particular, design of such software must start with writing down the exact grammar for the desired input language, determination of its complexity class and the appropriate class of the recognizer automaton for this language; the developers should then implement this automaton. Designers should take every effort to reduce the complexity class of the input language, preferring regular languages to context-free, context-free to context-sensitive, and so on; computational complexity exposed to inputs should be considered as another kind of "privilege" to be minimized away from attackers' reach.

LangSec argues that failure to construct input-handling software in this fashion is the root cause for the majority of security vulnerabilities exploited via crafted inputs. The essence of all such exploitation scenarios is that crafted inputs drive vulnerable code into unexpected state and through unexpected computation—in a process not unlike bytecode driving a virtual machine (see [2] for an informal overview, [3] for a formalism sketch). In a nutshell, crafted input acts as a *program* executed by the vulnerable software module.

Thus, in the LangSec view, *input validation* bears a striking similarity to *program verification* tasks, aimed at precluding inputs from driving their handling code into unexpected state and through unexpected computation. Such verification can only be achieved if the input-parsing code is constructed to verifiably reject any inputs outside of a formal-language definition (grammar) of valid or expected inputs.[1]

Program verification is notoriously hard. However, LangSec points out that in the special case of input language recognizer programs, the simpler the input language, the more constrained computation model its recognizer needs—and the more approachable the verification of such a recognizer becomes. Thus LangSec offers a way to apply verification where it has the biggest security impact—at communication boundaries, to rigorously defined input validation—while at the same time minimizing verification's costs by severely reducing the computational power of the execution environment of the program to be verified. *Whereas traditional verification problems implicitly assume that the underlying computational model of the code they target cannot be substantially simplified, LangSec posits that such simplification can and should be considered for input-parsing routines—as an important step toward security assurance.*

This paper seeks to demonstrate the scalability gain in the size of verification tasks from a restricted computational model. It introduces a sub-Turing programming language, *Crema*, and shows that it allows for easier verification and run-time monitoring through reduction of the verification problems' state-space growth. Since valid or expected input languages should not require full Turing-complete power of their recognizers, Crema could become a useful research benchmark for implementing verifiable input-validation modules in the LangSec paradigm.

A. Contributions

We describe a computational model that is less expensive to verify than the standard Turing-complete abstraction that underlies the development of general-purpose software. We

[1] Sometimes software may opt to process inputs that are *not* valid according to a protocol standard but are nevertheless expected, due to a long-time consensus. However, *all* such idiomatic inputs should be precisely specified as a part of the input language. Informal approaches such as attempts to "patch" invalid inputs by rewriting them have invariably led to security disasters, in which attackers co-opted the rewriting functionality (e.g., [4]).

DOI 10.1109/SPW.2015.30

analyze the benefits of a corresponding reduction in state-space search time of verification tasks for input validation of SMTP, a key Internet protocol. We also point out how our model helps to avoid the undecidability traps of which LangSec warns. An open-source, general-purpose programming language, *Crema*, is introduced that targets LLVM and can be used to develop embedded parsers that can be monitored with high efficiency and verified with higher assurance. Finally, we study this model empirically by applying it to verification of the *qmail* [5] parsing code compared to a Crema parser and determining the reductions in the state-space of the respective verification tasks.

II. Problem Specification

A. Background

The following sub-sections provide a brief background to the concepts built-upon in this work.

1) Language-theoretic Security: Language-theoretic Security (LangSec) sketches a unified view of software exploitation, with the concept of a "weird machine": an *ad hoc*, emergent virtual machine that converts input data into execution flow with unexpected states and state transitions [3]. From the LangSec perspective, return-oriented programming (ROP) is an example in which a *"weird machine"* is constructed by mining an existing code base such as *glibc* for executable "gadgets" (ending in `RET` instructions); the chain of faked stack frames of the exploit payload acts as its *program*, executing on the emergent machine composed of these gadgets (in addition to the memory-corrupting bug that overwrites the stack with this program). On a closer look, the addresses of these gadgets in memory form assembly-level byte-codes (op-codes) that drive the execution of the weird machine to unintended states in the control-flow graph (CFG). This construct has been described as early as 2001 by [6], [7]; its expressive power has been definitively described in terms of Turing-completeness in [8], finally shifting the academic security's threat model from the behind-the-times and narrow "malicious code" to "malicious computation" (see also [9]).

2) Software Tools for Our Case Study:

LLVM and KLEE: The LLVM compiler framework [10] is a tool-chain of modular components to analyze, optimize, compile, and execute programs via a standardized byte-code intermediate-representation (IR). Front-ends parse an input language, construct an abstract syntax tree (AST), and emit LLVM IR, which then can leverage the existing optimization passes, a cross-platform just-in-time compiler, and static analysis tools to allow for rapid compiler development.

Once an input program has been converted to the IR, there are a number of tools and libraries that can then be used to optimize the IR for faster execution or smaller memory footprint. There also exist a number of tools designed to aid in the analysis and verification of input programs, one of which is KLEE [11], a tool to symbolically execute input programs to search for test-cases that will lead to an error or crash. KLEE attempts to exercise all possible branches of the input program's CFG by substituting a symbolic value for each branch condition and continuing down both branches. If a certain path causes an error or program crash, it will use a constraint-solver on the symbolic branch conditions leading to the error in order to generate a concrete input test-case. The scope of these conditions must be carefully limited by human-provided hints to prevent the solver from crashing or running for an unreasonable/unbounded amount of time. This is of greater importance in programs that contain unbounded loops as the KLEE engine at times thinks each iteration of the unbounded loop is a new state, thus running the verification endlessly.

Due to the Halting Problem's undecidability for Turing-complete languages, KLEE cannot determine whether or not a certain program will terminate on a given input. In order to work around this, KLEE will execute the program symbolically for a preset amount of time (by default three minutes) and then terminate, thus there are some input programs that KLEE cannot exercise in entirety.

Qmail: a resilient secure Mail Transfer Agent: D.J. Bernstein's *qmail* [12] is a mail-transport agent (MTA) designed with security as a core requirement. The author had offered a reward for anyone who can report a security vulnerability in qmail in 1997; the only time the bounty was claimed was in 2009. In describing the lessons learned during the ten years of the MTA's development, Bernstein highlights the dangers of parsing input; accordingly, he worked to keep *qmail*'s internal file formats as simple as possible. Moreover, he recognized the SMTP parser as being the highest risk of compromise, and isolated it in a separate, untrusted process. Qmail's lessons remain highly relevant for today's high-profile input-related vulnerabilities such as Heartbleed, GnuTLS Hello overflow, MS SChannel, BERserk, and others.

B. LangSec-inspired Challenges

LangSec highlights the risks involved with parsing input into a program's internal type-system and demonstrates how a poorly designed or implemented parser creates a risk of unintended computations. The theory goes further and calls for input languages to be as restricted as possible, urging programmers to utilize the minimum amount of computational expressiveness necessary to validate inputs. This paper specifically looks at the challenges faced by tools designed to verify or automatically test program implementations in order to detect security risks before the software is put into production. Even with a strict input grammar and a well-designed parser, implementation flaws can still undermine the security of the overall program. Without fast and effective program analysis tools, those implementation flaws can persist into the production application.

It is in this light that Crema was devised, and this paper will compare verifiability of a restricted parser with that of a well-designed parser that has withstood many years of security scrutiny. By increasing the ability for automated testing and verification tools to detect implementation flaws, especially those in input parser, Crema aims to improve software security by putting a tool into the hands of software developers to employ LangSec theories transparently.

III. Solution

A. Crema Programming Language

As part of this research effort, we developed Crema, an open-source programming language environment targeting the LLVM tool-chain, with the hopes of demonstrating the benefits of a computationally restricted environment.

1) Motivation: The point of Crema is to implement certain development tasks with language-guaranteed termination and thus greatly ease verification of security-critical program elements (such as input recognizers/parsers). In particular, these tasks include input validation at communication boundaries, where verifying the correctness of the input recognizer's implementation is paramount to security. As most input-handling programs are intended by their developers to be *transducers*, performing computations on input and generating a resultant output, termination is not a roadblock to a majority of development projects. Clearly, there are a number of special exceptions to this pattern such as: operating system scheduling loops, event-handling loops, server listen loops, and read-evaluate-print loops (REPL) that interact with a user for an undetermined period of time (user-driven I/O).

It is, however the opinion of the authors that such an environment benefits a number of common programming tasks that notoriously produce vulnerabilities. As the LLVM IR resulting from a Crema program can be easily linked with any programs that can be compiled with LLVM, it is possible to develop the security-critical elements of a program, such as those routines that parse input in Crema, and verify them with higher assurance. Many development tasks for new efforts could be wholly written in Crema or a Crema-like restricted language to take advantage of the security benefits automatically provided.

2) Theoretical Underpinnings: The programming environment presented in this paper is based on the classic Turing machine described in [13], however the transition function (δ), is limited in such a way that it cannot return to an already-visited state:

$$\delta : (Q \setminus F) \times \Gamma \rightarrow Q' \times \Gamma \times \{L, R\}$$

where: Q is the finite set of states, F is the set of terminating states, Γ is the symbol alphabet, $\{L, R\}$ denote moving the tape reader head left or right, and Q' is the new set of states $Q' : Q \setminus q_c$ where q_c is the current state. Due to this limitation, the modified Turing machine will always terminate once all states in Q have been exhausted if not earlier. The state-space to search in order to verify δ has an upper bound of the number of states, thus a time-limit is not needed to determine if the verification is complete and exhaustive.

Just-in-time Function Inliner & Loop Unroller: The above model is highly restrictive and translates into a programming language with limited practical use. With the aforementioned limitations, looping and function calls are impossible to represent, as a `RET` in the function or a branch in the loop condition must return to a previously-visited state. In order to retain the benefits of the model, while still allowing function and (bounded) loop semantics, a JIT preprocessor is presented. This preprocessor is similar to the unrolling presented in [14].

In the aforementioned transition function, a set of states S is grouped into a named "model function", named with a unique symbol: $m_n \in \Gamma$. When the execution reaches said symbol, the model function is duplicated and inserted into Q, yielding $Q' : Q \cup S$. This process can be imagined as a mapping application of a replacement of a duplicate of the model function across all the unique function symbols in the program. When the transition function is loaded into the modified Turing machine, each function call will be inlined with a duplicate set of states and transitions representing the semantics of the function body.

A set of states can also be designated as a loop body and defined with a parameterized variable for the number of iterations to unroll. The start of each loop is denoted with a unique symbol. As the modified machine is executing, when the start of a loop symbol is reached, the JIT loop unroller will interrupt the machine and create n duplicate groups of the loop body (and optionally in-lining any functions within the loop), one for each loop iteration. The upper bound on the number of iterations (n) must be known at the time the loop begins (Walther recursion [15]), and thus must be a function of the inputs read from the tape and constant values. These new duplicate states are now able to be reached at most one time as per the above machine specification.

With this JIT inliner and the loop unroller, our modified Turing machine can emulate many of the semantics found in popular general-purpose programming languages and their associated run-time environments, excepting the unbounded loops or loops cannot have the number of iterations calculated *a priori*. In the following subsection, an equivalent programming language is described in detail and in Section IV is used to empirically contrast the state-space growths of a Turing-complete programming environment versus a more restricted model.

3) Language: Crema was designed to be a limited programming language that provides a restricted (but still practically useful) computational environment, while still having

⟨*integer*⟩ ::= ‘`char`’
| ‘`int`’
| ‘`uint`’

⟨*boolean*⟩ ::= ‘`true`’
| ‘`false`’

⟨*type*⟩ ::= void
| integer
| ‘`double`’
| boolean

⟨*string*⟩ ::= ‘`char`’[n]

Figure 1. Crema Types

```
int hundred[] = crema_seq(1, 100)

foreach(hundred as i) {
  int_print(i)
  str_print(" ")
  if (i % 3 == 0) {
    str_print("Fizz")
  }
  if (i % 5 == 0) {
    str_print("Buzz")
  }

  str_println(" ")
}
```

Figure 2. Sample Crema Program: “FizzBuzz”

a minimal learning-curve for software developers[2]. It is a weakly-typed (see Figure 1) procedural language supporting implicit up-casting of the below-defined “larger” types[3]:

$$\begin{cases} int < bool \\ double < bool \\ char < int < double \end{cases}$$

Crema has a similar syntax to Ruby or C, with the exception of not allowing unbounded loops or non-terminating recursion (or co-recursion). A BNF representation of the grammar can be found in Section III-A4. The core looping construct in Crema is the `foreach` instruction, which iterates through a list and performs the desired computations on each element of the list. As part of the Crema standard library, a `crema_seq(start, end)` function is provided to generate a sequence of consecutive numbers in a list for looping a certain number of times. While these constructs can be found in other programming languages (`seq()` in R, and `foreach` in PHP or LISP), what is unique about Crema is that it does *not* support unbounded looping constructs such as `while` in C-like languages of `loop` in LISP-like languages. An example Crema program, a slight modification to the widely-used interview question “FizzBuzz” [16], is presented in Figure 2 to provide the reader with an example of a program written for a restricted environment; the abbreviated LLVM assembly is shown in Appendix D.

The Crema compiler (`cremacc`) processes these input programs and converts them to LLVM IR byte-code, which then is optionally compiled to native machine code for the current platform. If the program is kept in the IR format, it is easily portable to other LLVM-support platforms and can be used as input for the existing tools that target the LLVM tool-chain.

4) Crema Grammar:

⟨*program*⟩ ::= ⟨*statements*⟩
| ⟨*empty*⟩

⟨*block*⟩ ::= ‘{’ ⟨*statements*⟩ ‘}’
| ‘{’ ‘}’

⟨*statements*⟩ ::= ⟨*statement*⟩
| ⟨*statements*⟩ ⟨*statement*⟩

⟨*statement*⟩ ::= ⟨*var_decl*⟩
| ⟨*struct_decl*⟩
| ⟨*func_decl*⟩
| ⟨*assignment*⟩
| ⟨*conditional*⟩
| ⟨*loop*⟩
| ⟨*return*⟩

⟨*var_decl*⟩ ::= ⟨*type*⟩ ⟨*identifier*⟩
| ⟨*type*⟩ ⟨*identifier*⟩ ‘=’ ⟨*expression*⟩
| ‘`struct`’ ⟨*identifier*⟩ ⟨*identifier*⟩
| ‘`struct`’ ⟨*identifier*⟩ ⟨*identifier*⟩ ‘=’ ⟨*identifier*⟩
| ⟨*list_decl*⟩

⟨*struct_decl*⟩ ::= ‘`struct`’ ⟨*identifier*⟩ ‘{’ ⟨*var_decls*⟩ ‘}’

⟨*func_decl*⟩ ::= ⟨*def*⟩ ⟨*type*⟩ ⟨*identifier*⟩ ‘(’ ⟨*func_decl_arg_list*⟩ ‘)’ ⟨*block*⟩
| ⟨*def*⟩ ⟨*type*⟩ ‘[’ ‘]’ ⟨*identifier*⟩ ‘(’ ⟨*func_decl_arg_list*⟩ ‘)’ ⟨*block*⟩
| ‘`extern`’ ⟨*def*⟩ ⟨*type*⟩ ⟨*identifier*⟩ ‘(’ ⟨*func_decl_arg_list*⟩ ‘)’

[2]Crema is under heavy development and, as such, syntax and programming APIs may change over time.

[3]E.g., `double d = 1` is a valid Crema statement, but `int i = 1.0` is not and must use an explicit conversion

| ‘extern’ ⟨*def*⟩ ⟨*type*⟩ ‘[’ ‘]’ ⟨*identifier*⟩ ‘(’ ⟨*func_decl_arg_list*⟩ ‘)’

⟨*assignment*⟩ ::= ⟨*identifier*⟩ ‘=’ ⟨*expression*⟩
| ⟨*list_access*⟩ ‘=’ ⟨*expression*⟩
| ⟨*struct*⟩ ‘=’ ⟨*expression*⟩

⟨*conditional*⟩ ::= ‘if’ ‘(’ ⟨*expression*⟩ ‘)’ ⟨*block*⟩
| ‘if’ ‘(’ ⟨*expression*⟩ ‘)’ ⟨*block*⟩ ‘else’ ⟨*block*⟩
| ‘if’ ‘(’ ⟨*expression*⟩ ‘)’ ⟨*block*⟩ ‘else’ ⟨*conditional*⟩

⟨*loop*⟩ ::= ‘foreach’ ‘(’ ⟨*identifier*⟩ ‘as’ ⟨*identifier*⟩ ‘)’ ⟨*block*⟩

⟨*return*⟩ ::= ‘return’ ⟨*expression*⟩

⟨*type*⟩ ::= ‘double’
| ‘int’
| ‘str’
| ‘void’
| ‘bool’

⟨*var_decls*⟩ ::= ⟨*var_decl*⟩
| ⟨*var_decls*⟩ ‘,’ ⟨*var_decl*⟩
| ⟨*empty*⟩

⟨*def*⟩ ::= ‘def’

⟨*func_decl_arg_list*⟩ ::= ⟨*var_decl*⟩
| ⟨*func_decl_arg_list*⟩ ‘,’ ⟨*var_decl*⟩
| ⟨*empty*⟩

⟨*list_decl*⟩ ::= ⟨*type*⟩ ⟨*identifier*⟩ ‘{’ ‘}’
| ⟨*type*⟩ ⟨*identifier*⟩ ‘{’ ‘}’ ‘=’ ⟨*expression*⟩

⟨*expression*⟩ ::= ⟨*term*⟩
| ⟨*expression*⟩ ⟨*bitwise*⟩ ⟨*term*⟩
| ⟨*expression*⟩ ‘+’ ⟨*term*⟩
| ⟨*expression*⟩ ‘−’ ⟨*term*⟩

⟨*term*⟩ ::= ⟨*factor*⟩
| ⟨*term*⟩ ‘*’ ⟨*factor*⟩
| ⟨*term*⟩ ‘/’ ⟨*factor*⟩
| ⟨*term*⟩ ‘%’ ⟨*factor*⟩
| ⟨*term*⟩ ⟨*comparison*⟩ ⟨*factor*⟩

⟨*comparison*⟩ ::= ‘==’
| ‘!=’
| ‘>’
| ‘<’
| ‘>=’
| ‘<=’
| ‘&&’
| ‘||’

⟨*bitwise*⟩ ::= ‘&’
| ‘^’
| ‘|’

⟨*factor*⟩ ::= ⟨*var_access*⟩
| ⟨*list*⟩
| ⟨*value*⟩
| ⟨*identifier*⟩ ‘(’ ⟨*func_call_arg_list*⟩ ‘)’
| ‘(’ ⟨*expression*⟩ ‘)’
| ‘−’ ‘(’ ⟨*expression*⟩ ‘)’

⟨*var_access*⟩ ::= ⟨*identifier*⟩
| ⟨*list_access*⟩
| ⟨*struct*⟩
| ‘−’ ⟨*var_access*⟩

⟨*list_access*⟩ ::= ⟨*identifier*⟩ ‘[’ ⟨*expression*⟩ ‘]’
| ⟨*identifier*⟩ ‘[’ ‘]’

⟨*struct*⟩ ::= ⟨*identifier*⟩ ‘.’ ⟨*identifier*⟩

⟨*list*⟩ ::= ‘{’ ⟨*func_call_arg_list*⟩ ‘}’

⟨*func_call_arg_list*⟩ ::= ⟨*expression*⟩
| ⟨*func_call_arg_list*⟩ ‘,’ ⟨*expression*⟩
| ⟨*empty*⟩

⟨*value*⟩ ::= ⟨*numeric*⟩
| ⟨*string*⟩
| ‘true’
| ‘false’

⟨*numeric*⟩ ::= ⟨*double*⟩
| ‘−’ ⟨*double*⟩
| ⟨*int*⟩
| ‘−’ ⟨*int*⟩

IV. Experimental Evaluation

To empirically measure the benefits of this restricted programming environment with respect to verification, the *qmail-smtpd* parser code was executed symbolically with KLEE while recording metrics of the state-space growth in two different testing scenarios. The *qmail* parser is an example of very well-designed and securely developed code

written in a fully Turing-complete environment. A simple, yet similar parser was developed in the restricted Crema programming language in order to compare the code-coverage and state-space explosion when tested by KLEE. KLEE was used to explore the state-space growth and detect crash cases, not to verify certain properties about the program, thus no source code annotations are used.

Why qmail: Our selection of *qmail* to test a measure to ease verification of programs designed with LangSec in mind is not incidental. Firstly, *qmail* design stresses the perils of parsing and the necessity to isolate parsers of data coming from the network with all possible OS means. This is entirely consistent with the LangSec view and provides us with the parser cleanly separated from the rest of the MTA logic. Thus, using *qmail* removes the challenge of having to draw arbitrary boundaries between the code that validates inputs (and thus must be verified to recognize and accept exactly the inputs specified as valid) and the code that is written based the assumption that such validation has occurred—with *qmail*, this separation that LangSec demands explicitly is approached very closely if not perfectly.

Secondly, *qmail*'s input parsing tasks are clearly representative of the real challenges faced by programs that must take untrusted input from the Internet. Despite being "simple", SMTP has seen a number of notoriously unsafe implementations before *qmail*, with input-handling bugs that became iconic in their class. At the same time, as an MTA agent for an already widely deployed protocol, *qmail* is under pressure to accommodate existing input variations and dialects rather than being at liberty to merely subset them and discard all inputs non-compliant with the chosen subset. Thus, *qmail*'s parsing is representative of the complexity associated with handling a broadly deployed "legacy" protocol.

Test I: In the first test, the *qmail-smtpd* parser is removed from the unbounded input loop, and the parser function is passed a symbolic input of increasing length. It is then exercised normally by KLEE, as in the verification of a typical program or in an AEG tool's search. KLEE is instructed (with the `--only-output-states-covering-new` argument) to keep separate statistics for the total number of paths explored and the number of unique paths without revisiting previously exercised states. This scenario is not guaranteed to exercise the entire CFG and may miss certain states due to the state-space explosion. The latter statistic is used to model the restricted transition function described in Section III-A2. The difference in growth rates of total and "trimmed" paths was measured and is detailed in Section IV-A.

Test II: The *qmail-smtpd* program has two significant functions that handle input parsing (`commands` and `addrparse`)[4]. The initial parsing routine is the simpler of the two. It reads a string from an input buffer and attempts to split the string into a command and an optional argument. The command portion is used to index a table of function handlers. Specific commands will trigger the second parsing routine to parse the argument. For the sake of simplicity, the entire input was defined to be symbolic, but in order to target the second parsing routine the command portion was fixed (via `klee_assume()`). The argument portion was defined to be at most eight bytes long in order to maximize code coverage while restricting state space growth.

In the second test, the *qmail-smtpd* parser code is modified to simulate execution in a restricted computation environment equivalent to the Crema execution mode. Unbounded loops are removed, and tests are performed with fixed length symbolic inputs to limit the number of iterations performed by input parsing loops (approximating the *transducer*-like environment explained in Section III-A1). In this environment, the state-space is drastically reduced by the lack of unbounded loops, and verification is faster, thus able to practically handle larger code-bases for checking.

In this latter test, both the modified and unmodified *qmail-smtpd* parser was symbolically executed multiple times with increasing limits on execution time. The number of paths explored, instruction coverage, and average program states, among other metrics, were all recorded for each test[5]. This test aims to show the benefits of programming in a language with restricted computational expressiveness, such as described in Section III-A.

Test III: This final test used KLEE to exercise a prototype parser written in Crema designed to operate in a similar fashion to the *qmail* parser. The goal of this test case was to explore state-space growth in a restricted and sub-Turing programming environment versus a well-designed, but Turing-complete parser. The source code for the parser (edited slightly for brevity and clarity) can be found in Appendix B along with the corresponding qmail commands parser functionality in Appendix C (with the infinite loop commented out). Both of these parsers were executed symbolically with KLEE in order to compare how a very well-written and security-conscious parser written in C would compare with a prototype Crema parser vis-à-vis state-space growth. The results for this test are discussed in Section IV-A.

A. Results and Discussion

The results of these experiments are outlined below for both testing scenarios. It should be noted that, due to the complexity of the C standard library, the instruction and branch code coverage of KLEE during execution is greatly diminished. As a control for baseline KLEE coverage, the

[4] `int commands(ss, c)` (line 9 in 'commands.c') and `int addrparse(arg)` (line 140 in 'qmail-smtpd.c') in qmail-1.03

[5] The design of the *qmail* parser is such that many paths cannot be explored without a symbolic input string of a certain length. This is due to argument parsing, and the KLEE run-times grew too quickly to fully explore the parser with a longer input string

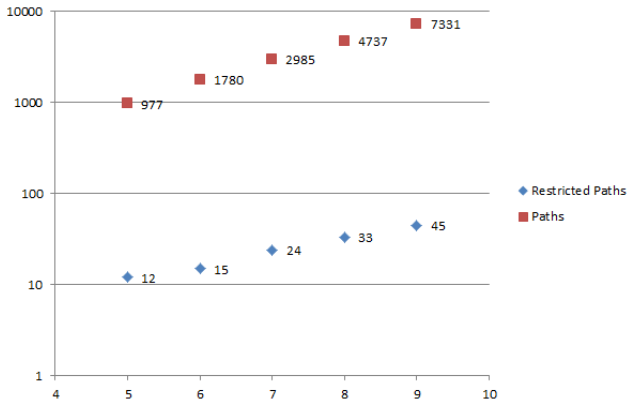

Figure 3. Explored Paths and Trimmed Paths vs. Symbolic Input Length

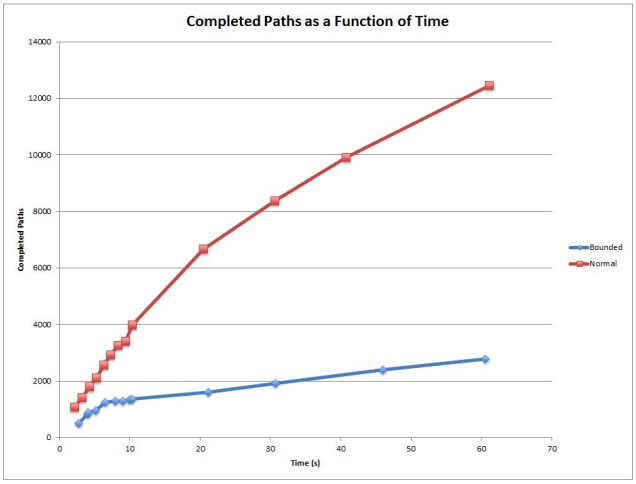

Figure 4. Completed Paths in Time-limit as Function of Time

simple "Hello, World!" program was written in C (Figure 5) and exercised by KLEE. The resulting code coverage by KLEE was only 17% instruction coverage and 12% branch coverage.

Test I Results: The results comparing the difference in growth between the total paths and trimmed paths are shown in Table I. The length of the symbolic input string used as an input parameter for the qmail parser was varied to exercise an increasing amount of the code-base. Input lengths greater than 9 characters executed too long to return results. The percentage of total code exercised is shown in the table, as well as the total run-time for the KLEE symbolic engine. The number of paths explored quantify the growth in the state-space that the KLEE engine must explore to fully exercise the code reachable based on the symbolic input format. A comparison between the explored paths and trimmed paths as a function of symbolic input length is shown in Fig. 3.

Test II Results: The results from Test II (Table II) highlight the verification benefits of restricting the computational expressiveness of the program source code, a goal of the language described in Section III-A. This test shows the improved code coverage and decreased number of states that must be checked when compared with a standard implementation. These results more closely mirror the program after the function inliner and loop unroller described in Section III-A2. In Fig. 4, a plot provides the number of paths the KLEE symbolic engine had to explore in the provided time-limit before it was halted. In this test case, the percentage of the program that KLEE was able to explore (instruction coverage) in the set time-limit was slightly higher for the bounded program than the unbounded parser.

Test III Results: The results from Test III, shown in Table III, highlight the state-space size growth advantages of the restricted computational environment implemented via a Crema program. The number of states that KLEE explored is significantly reduced in the Crema parser when compared to similar C code from *qmail*, helping to empirically reaffirm what is intuitively expected—that verification and exercising restricted computational models is easier than fully Turing-complete environments. When the code in Appendix C was not modified to comment out the unbounded outer loop, KLEE would run apparently without making progress until the default 3 minute timeout is reached. Examining the code coverage between the two parsers also shows the Crema program having a considerably higher percentage instruction coverage than when KLEE exercised *qmail*.

B. Results Interpretation

As shown, the restricted environment results in a clear reduction of the state-space explosion.

Experience suggests that this reduction is significant. Recently, NICTA was able to formally verify the seL4 micro-kernel [17] that was approx. 10,000 lines of high-level code (automatically and provably translated into approx. 10,000 lines of C code). Larger code-bases have resisted verification due to the state-space explosion. The results from our experiment suggest significant reduction in the state-space during verification (e.g., by a factor of 4 for completed paths), greatly increasing the possible code-base size possible to verify in this model.

We note that this reduction resulted for code that is well-architected and written with security in mind, but unmodified; for code that specifically targets such reductions, the gains are likely to be significantly larger.

Many components of existing, unverified software projects can be modeled in such a restricted fashion and then verified. Components of the Linux kernel (for example) could be modeled in a restricted environment (avoiding synchronization and critical sections) and then verified for correctness, improving the assurances provided by the platform.

Table I
QMAIL STATE-SPACE EXPLOSION REDUCTION IN RESTRICTED ENVIRONMENT

Symbolic Input Size (characters)	Instruction Coverage (%)	Explored Paths	Trimmed KLEE Paths	KLEE Run-time (s)
5	31.37	997	12	1.12
6	31.59	1780	15	2.03
7	33.13	2985	24	6.23
8	33.97	4737	33	53.27
9	34.62	7331	45	540.2

Table II
COMPLETED PATHS IN TIME-LIMIT FOR BOUNDED AND UNBOUNDED PARSER

Time Limit (s)	Completed Paths in Unbounded Parser	Completed Paths in Bounded Parser
2	1068	511
3	1412	847
4	1803	880
5	2115	966
6	2573	1246
7	2921	1295
8	3263	1283
9	3413	1344
10	3977	1363
20	6679	1613
30	8365	1925
40	9899	2386
60	12451	2783

Table III
QMAIL C PARSER COMPARED TO CREMA PARSER

Parser	Execution Time (s)	Maximum States	Instruction Coverage (%)	Branch Coverage (%)
Qmail C	31.50	678	44.47	33.96
Crema	28.67	76	61.97	37.74

V. RELATED WORK

A. Modeling Execution Events as an Input Language

In [18], a reference monitor is defined as an automaton that recognizes a "language of events". This model of reference monitors interprets the events/actions performed by a monitored program as a stream of symbols; the reference monitor can reject certain "inputs" as incorrect, detecting compromises of the monitored process.

Due to the undecidability of the Halting Problem, for a general process, the monitor is restricted to only checking "input prefixes". The reference monitor can detect if a process is starting to misbehave based on certain patterns of events, however cannot recognize more complex input language grammar classes. With the restricted model, a reference monitor could potentially recognize more complex languages of inputs and possibly roll back events performed by a compromised process after it terminated, recovering trustworthy state.

Presently, a reference monitor is limited by two factors: the undecidability cliff and the fact that the reference monitor is operating with the same level of computational expressiveness as the process it is monitoring. Similar to the cat-and-mouse game with malware and anti-virus operating at the same level of privilege, the fact that the reference monitor is not more powerful than the monitored process weakens its abilities to recognize compromise.

Automated Exploit Generation (AEG): AEG [19] and its precursor [20] is an effort to automatically find exploitable vulnerabilities and to generate proof-of-concept exploits for them, such as inputs that cause the computation to be diverted to afford the attacker full control of the target. AEG started with automatic generation of crafted input-programs (payloads) for the classic execution model of stack buffer overflows (described in [21]). This model is nearly extinct in modern desktop software thanks to the defensive measures such as DEP, ASLR, EMET, etc., which co-evolved with the state-of-the-art offensive methods since the 1990s; however, this model is alive and well on the ubiquitous micro-controller firmware that appear posed to

drive the so-called "Internet of Things".[6]

In the later work [19], the authors described AEG as a *program verification task but with a twist*—the twist being that typical safety properties are replaced with finding a certain unexpected kind of a program execution path when subjected to crafted inputs. From the LangSec perspective, an AEG algorithm actually finds both a description of an input-driven "weird machine" and the program that drives it to some definition of an undeniably unexpected (a.k.a. " malicious") computation, an exploit.[7]

B. Common challenge for AEG and Program Verification: the State Explosion

Both problems, verification and AEG, are related and the research in one field impacts the other [22]. In verification, the verifier is aiming to prove that there are no unintended states in the program that are reachable (in many cases, in comparison to a specification or formal model), whereas in AEG, the generator is attempting to prove that there exists a reachable unintended/error state.[8]

As these problems are closely related, they both are impacted by the state-space explosion when mapping the CFG for Turing-complete software programs; additionally the Halting Problem introduces the issue of determining whether to continue searching or stop. Computational complexity of general-purpose programming environment impedes automating solutions of both problems.

The general-purpose programming languages in use today provide more computational expressiveness than is needed to perform most input-validation software tasks. The gap between the needed and more easily verifiable "power" and what is available is a source of security risk [23]. This gap, known in LangSec as the "undecidability cliff", prevents both formal verification and AEG tools from analyzing completely a general, non-trivial input program [24].

VI. Conclusion

The authors posit that, for most common input-handling purposes, the fully expressive, Turing-complete environment is overly powerful and carries a significant (and realized) risk of compromise. The majority of programming languages aim for Turing-completeness, then focus on syntax and library functions. The limited model and the Crema language described in this paper have been specifically designed to not aim for Turing-completeness, but rather with a view towards practically and safety, to allow program-verification tools to explore the resulting programs' state space. By providing a restricted execution model and showing the verification benefits thereof, future work can explore existing code bases and perform analyses that identify components or subsystems that could be modeled in this restricted environment, allowing for verification of code-bases currently too large to check.

[6]Occasionally referred in the security community as "The Internet of Things that Explode", for this and other reasons.

[7]As the noted security researcher Felix 'FX' Lindner put it, "You can't argue with a root shell."

[8]Quoting [19], *Casting AEG in a verification framework ensures AEG techniques are based on a firm theoretic foundation. The verification-based approach guarantees sound analysis, and automatically generating an exploit provides proof that the reported bug is security-critical.*

Acknowledgments

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA)[9]. The authors would like to thank Julien Vanegue and Thomas Dullien for their input on the state-space explosion problems prevalent in verification/AEG and Sergey Bratus for his comments and suggestions on earlier drafts of this paper.

References

[1] L. Sassaman *et al.*, "Security applications of formal language theory," *IEEE Systems Journal*, vol. 7, no. 3, 2013.

[2] S. Bratus *et al.*, "Exploit programming: from buffer overflows to weird machines and theory of computation," *USENIX ;login:*, 2011.

[3] J. Vanegue, "The weird machines in proof-carrying code," in *Proc. First Annual Langsec Workshop*, May 2014.

[4] E. Nava and D. Lindsay. (2010, Apr.) Abusing Internet Explorer 8's XSS filters. [Online]. Available: http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf

[5] D. J. Bernstein, "Some thoughts on security after ten years of qmail 1.0," in *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, ser. CSAW '07. New York, NY, USA: ACM, 2007, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/1314466.1314467

[6] G. Richarte. (2000, Oct.) Re: Future of buffer overflows. [Online]. Available: http://seclists.org/bugtraq/2000/Nov/32

[7] Nergal. (2001, Dec.) The advanced return-into-lib(c) exploits. [Online]. Available: http://phrack.org/issues/58/4.html

[8] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. ACM, 2007.

[9] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 383–398. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855768.1855792

[10] C. Lattner. (2015) The LLVM compiler infrastructure. [Online]. Available: http://llvm.org/

[9]The views, opinions, and/or findings contained in this article are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

[11] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of USENIX OSDI 2008*, San Diego, CA, Nov 2008.

[12] D. J. Bernstein. (2013) qmail. [Online]. Available: http://cr.yp.to/qmail.html

[13] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[14] A. Lal, S. Qadeer, and S. Lahiri, "Corral: A whole-program analyzer for boogie."

[15] C. Walther, "Security applications of formal language theory," *Artificial Intelligence*, vol. 70, no. 1, 1994.

[16] I. Ghory. (2007, Jan.) Using FizzBuzz to find developers who grok coding. [Online]. Available: http://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/

[17] G. Klein, "Operating system verification — an overview," *Sādhanā*, vol. 34, no. 1, pp. 27–69, feb 2009.

[18] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, pp. 30–50, Feb. 2000. [Online]. Available: http://doi.acm.org/10.1145/353323.353382

[19] T. Avgerinos *et al.*, "Automatic exploit generation," *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, 2014.

[20] S. Heelan, "Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities," Master's thesis, University of Oxford, Oxford, UK, 2009.

[21] A. One. (1996, Aug.) Smashing the stack for fun and profit. [Online]. Available: http://phrack.org/issues/49/14.html

[22] J. Vanegue, "The automated exploitation grand challenge," presented at H2HC Conference, Oct 2013.

[23] L. Sassaman *et al.*, "The halting problems of network stack insecurity," *USENIX ;login:*, vol. 36, no. 6, 2011.

[24] S. Bratus and F. Lindner, "Information security war room," presented at the Proc. USENIX Security, 2014.

APPENDIX A.
HELLO WORLD CODE COVERAGE SAMPLE

```
#include <stdio.h>

int main(int argc, char ** argv)
{
  printf(``Hello World\n'');
  return 0;
}
```

Figure 5. "Hello, World" C Sample Program

APPENDIX B.
PROTOTYPE CREMA PARSER SOURCE (ABBREVIATED)

```
def int commands(string c)
{
        string cmd
        string arg
        int len = str_len(c)
        int itr = 0
        int max_input_itr[] =
        crema_seq(0, len)
        foreach(max_input_itr as itr) {
            if (c[itr] == '\n') {
               if (itr > 0) {
                       int i = itr - 1
                       cmd = str_substr(
                       c, 0, i)
                       break
               }
            }
        }
        len = str_len(c)
        itr = str_chr(cmd, ' ')
        arg = str_substr(cmd, itr, 0)

        if (str_compare(cmd, "rcpt") == 1) {
           smtp_rcpt(arg)
        } else if (str_compare(
        cmd, "mail") == 1) {
           smtp_mail(arg)
        } else if (str_compare(
        cmd, "data") == 1) {
           smtp_data()
        } else if (str_compare(
        cmd, "quit") == 1) {
           smtp_quit()
        } else if (str_compare(
        cmd, "helo") == 1) {
           smtp_helo(arg)
        } else if (str_compare(
        cmd, "ehlo") == 1) {
           smtp_ehlo(arg)
        } else if (str_compare(
        cmd, "rset") == 1) {
           smtp_rset()
        } else if (str_compare(
        cmd, "help") == 1) {
           smtp_help()
        } else if (str_compare(
        cmd, "noop") == 1) {
           err_noop()
        } else if (str_compare(
        cmd, "vrfy") == 1) {
           err_vrfy()
        } else {
           err_unimpl()
        }

        return 0
}

int argc = prog_arg_count()
string command = prog_argument(1)

commands(command)
```

APPENDIX C.
STRIPPED-DOWN QMAIL PARSER SOURCE (ABBREVIATED)

```
int commands(ss,c)
```

```
char *ss;
struct commands *c;
{
  int i;
  char *arg;

  //  for (;;) {
    if (!stralloc_copys(&cmd,""))
        return -1;

    for (;;) {
      if (!stralloc_readyplus(&cmd,1))
        return -1;
      cmd.s[cmd.len] = ss[cmd.len];
      if (cmd.s[cmd.len] == '\0')
        return 0;
      if (cmd.s[cmd.len] == '\n')
        break;
      ++cmd.len;
    }

    if (cmd.len > 0)
        if (cmd.s[cmd.len - 1] == '\r')
                --cmd.len;

    cmd.s[cmd.len] = 0;

    i = str_chr(cmd.s,' ');
    arg = cmd.s + i;
    while (*arg == ' ') ++arg;
    cmd.s[i] = 0;

    for (i = 0;c[i].text;++i)
        if (case_equals(c[i].text,cmd.s))
                break;
    c[i].fun(arg);
    if (c[i].flush) c[i].flush();
    //  }
}

struct commands smtpcommands[] = {
    { "rcpt", smtp_rcpt, 0 }
    , { "mail", smtp_mail, 0 }
    , { "data", smtp_data, flush }
    , { "quit", smtp_quit, flush }
    , { "helo", smtp_helo, flush }
    , { "ehlo", smtp_ehlo, flush }
    , { "rset", smtp_rset, 0 }
    , { "help", smtp_help, flush }
    , { "noop", err_noop, flush }
    , { "vrfy", err_vrfyf, flush }
    , { 0, err_unimpl, flush }
} ;

int main(int argc, char ** argv)
{
    if (commands(
        argv[1], &smtpcommands) == 0)
            die_read();
    die_nomem();
}
```

APPENDIX D.
ABBREVIATED FIZZBUZZ LLVM IR

```
; ModuleID = 'Crema JIT'
target triple = "x86_64-pc-linux-gnu"

@hundred = internal global i8* undef
@loopItCnter = internal global i64 undef
@i = internal global i64 undef

define i64 @main(i64 %argc, i8** %argv) {
entry:
  call void @save_args(
        i64 %argc, i8** %argv)
  %0 = call i8* @crema_seq(i64 1, i64 100)
  store i8* %0, i8** @hundred
  br label %preblock

preblock:
  store i64 0, i64* @loopItCnter
  br label %bodyblock

bodyblock:
  %1 = load i8** @hundred
  %2 = load i64* @loopItCnter
  %3 = call i64
        @int_list_retrieve(i8* \%1, i64 \%2)
  store i64 %3, i64* @i
  %4 = load i64* @i
  call void @int_print(i64 %4)
  %5 = call i8* @str_create()
  call void @str_append(i8* %5, i8 32)
  call void @str_print(i8* %5)
  %6 = load i64* @i
  %7 = srem i64 %6, 3
  %8 = icmp eq i64 %7, 0
  br i1 %8, label %15, label %17

loopcondblock:
  %9 = load i64* @loopItCnter
  %10 = add i64 1, %9
  store i64 %10, i64* @loopItCnter
  %11 = load i8** @hundred
  %12 = call i64 @list_length(i8* %11)
  %13 = load i64* @loopItCnter
  %14 = icmp eq i64 %13, %12
  br i1 %14, label %termblock, label %bodyblock

; <label>:15
  %16 = call i8* @str_create()
  call void @str_append(i8* %16, i8 70)
  call void @str_append(i8* %16, i8 105)
  call void @str_append(i8* %16, i8 122)
  call void @str_append(i8* %16, i8 122)
  call void @str_print(i8* %16)
  br label %17

; <label>:17
  %18 = load i64* @i
  %19 = srem i64 %18, 5
  %20 = icmp eq i64 %19, 0
  br i1 %20, label %21, label %23

; <label>:21
  %22 = call i8* @str_create()
  call void @str_append(i8* %22, i8 66)
  call void @str_append(i8* %22, i8 117)
  call void @str_append(i8* %22, i8 122)
  call void @str_append(i8* %22, i8 122)
  call void @str_print(i8* %22)
  br label %23

; <label>:23
  %24 = call i8* @str_create()
  call void @str_append(i8* %24, i8 32)
  call void @str_println(i8* %24)
  br label %loopcondblock

termblock:
  ret i64 0
}
```