Featherweight VeriFast

Frédéric Vogels, Bart Jacobs, and Frank Piessens

KULeuven

Abstract. VeriFast is a verification tool based on separation logic which can be used to verify correctness properties of C and Java programs, going from memory safety and absence of race conditions to full functional correctness. While the tool has been applied to several real-world cases, VeriFast as of yet had no published theoretical foundation. This paper fills this lacuna by giving a full formalisation and soundness proof of a core subset of VeriFast. It is our belief that our novel approach results in a more elegant mathematical description than other formalisations.

1 Introduction

VeriFast [1, 2] is a static verification tool for C and Java, supports both single-threaded and multithreaded code, and has been used in several real-world applications [3, 4]. The tool requires the code to be annotated (e.g. preconditions and postconditions) using separation logic [5] formulae and ghost commands. Automatic generation of these annotations [6] is an orthogonal issue. In this paper, we present a formalisation of a subset of VeriFast with a proof of its soundness.

In our opinion, a formalisation should be more than a series of definitions; it should be accompanied by the rationale behind it. Explaining it clearly in limited space is quite a challenge as decisions and the motivations behind them reside on different levels of abstraction. Therefore, we have chosen to split the contents of this paper into two passes. The first (Sect. 2) is an informal build-up explanation, the second (Sect. 3, Sect. 4 and Sect. 5) gives the full formalisation.

The formalisation starts with the definition of a small imperative language (Sect. 2.1 and Sect. 4.1), for which we define the semantics, named the concrete execution (Sect. 2.2 and Sect. 4.2). For verification purposes, this semantics suffers from serious issues, which are solved by the symbolic execution (Sect. 2.3 and Sect. 4.3). Lastly, in Sect. 5, we prove that the symbolic execution is a reliable way to perform verification. Full definitions and proofs are available in the extended version [7] of this paper.

2 Rationale

In this section, we give an informal description of Featherweight VeriFast. Section 2.1 introduces the small imperative language, after which Sect. 2.2 defines its semantics, i.e. the concrete execution. This semantics has some shortcomings, which the symbolic execution addresses (Sect. 2.3).

2.1 The Small Imperative Language

The small imperative language (SIL) is a C-like language. It supports only one type of values, i.e. natural numbers (with no size limit) which are also used as memory addresses. We impose no limit on the size of the heap, so that any natural number represents a valid memory location. We defer its full definition until Sect. 4.1.

2.2 Concrete Execution

The concrete execution (CE) models execution of a SIL-program as it would happen on a machine. It is a function relating an input state directly to its end results. The program state is modelled as a store/heap pair, denoted $\langle s, h\rangle_c$. The store s is a function mapping variable names to values. The heap h is a multiset of *heap chunks* of which there are two kinds: an $\mathsf{mb}(\ell, n)$ chunk indicates that n memory cells were allocated, starting at location ℓ. The second chunk type, written $\ell \mapsto v$, represents the fact that the memory cell at location ℓ contains the value v. Its presence on the heap is required to read or write from that location, or failure ensues. Whenever $\mathsf{mb}(\ell, n)$ is present on the heap, so are corresponding $\ell \mapsto v_0, \ell + 1 \mapsto v_1, \ldots, \ell + n - 1 \mapsto v_{n-1}$ chunks. The reason for the existence of a separate mb chunk is that we need to know at runtime how many of the $\ell \mapsto v$ cells must be removed from the heap when freeing memory.

The CE can fail in multiple ways. For example, dereferencing a dangling pointer (i.e. reading from a memory location ℓ for which there is no $\ell \mapsto v$ chunk on the heap) is considered an error. The goal of verification is to detect beforehand if such errors could occur during CE. We say a program verifies if the verifier determines that no such errors will be encountered during any possible execution. The verifier is sound if it is right every time[1].

A complicating matter is the nondeterministic nature of the CE. This might seem strange as the CE is meant to faithfully mimic a machine's operation, which is inherently deterministic. The nondeterminism arises from having to take into account external factors such as user input: a program must succeed for any possible external input.

To keep our language semantics minimal, we introduce a single source of nondeterminism, namely memory allocation: a new memory block can be allocated anywhere in free memory, and no guarantees are made about the initial contents of the freshly allocated memory cells. Other sources of randomness can be built on top of this.

A deterministic CE would map each state to a single state. To allow nondeterminism, we generalize[2] its type signature to

$$\text{execute} : \mathit{Statement} \longrightarrow \mathit{ProgramState} \longrightarrow \mathcal{P}(\mathit{ProgramState})$$

[1] The verifier only has to be right when it says the program is error-free; in other words, a verifier which mistakenly says the program is incorrect is still sound.

[2] Note that execute will be generalized even further.

where $\mathcal{P}(A)$ denotes the powerset of A. For example, execution of the command $x := \mathbf{malloc(2)}$ in a state consisting of the zero store s_0 (which maps all variables to 0) and an empty heap, leads to the infinite set of states

$$\left\{\begin{array}{c} \langle s_0[x := 5], \{\mathsf{mb}(5,2), 5 \mapsto 9, 6 \mapsto 7\}\rangle_c \\ \langle s_0[x := 7], \{\mathsf{mb}(7,2), 7 \mapsto 0, 7 \mapsto 3\}\rangle_c \\ \vdots \end{array}\right\}$$

CE-trees can have both infinite breadth (due to the infinite heap and infinite value domain) and depth (endless recursion). In order to be able to verify programs, we need a way to cut those trees down to a finite size. For this reason, we will now introduce the *symbolic execution* (SE).

2.3 Symbolic Execution

The symbolic execution is a finite approximation of the concrete execution. We discuss this in a stepwise manner.

Assertions We solve the infinite depth issue by first extending every routine with a specification which it must obey. A routine call then becomes an atomic step which expects the current state to satisfy the routine's precondition after which it transforms the state nondeterministically into one which satisfies the postcondition. This takes care of the infinite depth but has worsened the infinite breadth situation. Let us worry about this later.

The routine contract needs to be written down in the assertion language which is based on separation logic (see Def. 7). The main difference is instead of using general existential quantification, we make use of a form of pattern matching. For example, a points-to assertion takes the form $e \mapsto ?x$, where $?x$ indicates that x will be bound to the value of the corresponding heap cell. The scope of these variables extends until the end of the contract they appear in.

$$\textbf{routine } \mathrm{inc}(x) \textbf{ req } x \mapsto ?v \textbf{ ens } x \mapsto v + 1 \;=\; t := [x]; t := t + 1; [x] := t$$

The inc routine increments the contents of a given heap cell by one. The variable v relates the postcondition with the precondition. For example, the call $\mathrm{inc}(p)$ would transform the program state $\langle s_0[p := 5], 5 \mapsto 8\rangle_c$ into $\langle s_0[p := 5], 5 \mapsto 9\rangle_c$.

Internally, a routine call is verified by first *consuming* the precondition and then *producing* the postcondition. Consumption consists of checking that all necessary conditions hold (if not, failure ensues), such as the presence of certain heap chunks. When a heap chunk is matched, the necessary variables are bound and the chunk is removed from the heap. Chunk production adds heap chunks to the heap and assigns values to unbound variables in a nondeterministic way.

Heap abstraction Let us consider (noncyclic) linked lists. For simplicity, we leave out the actual list items and assume a linked list merely consists of a series of heap cells pointing to each other, the chain eventually ended by a zero pointer.

A length routine which counts the number of jumps before 0 is reached needs to be able to work on lists of arbitrary length. The CE would have us work with an infinite number of states, each representing a different list in memory. We solve this problem by introducing custom chunks (also known as user defined predicates.)

$$\mathrm{list(p)} = \textbf{if } \mathrm{p} = 0 \textbf{ then true else } \mathrm{p} \mapsto \mathrm{q} \,\star\, \mathrm{list(q)}$$

where $\star$ denotes separating conjunction [5], indicating that $\mathrm{p} \mapsto \mathrm{q}$ and $\mathrm{list(q)}$ do not share heap cells. This way, a single chunk can represent all possible linked lists. For example, the single state[3] $\langle s_0, \{\mathrm{list}(5)\}\rangle_c$ represents $\langle s_0, \{5 \mapsto 0\}\rangle_c$, $\langle s_0, \{5 \mapsto 4, 4 \mapsto 0\}\rangle_c$, $\langle s_0, \{5 \mapsto 61, 61 \mapsto 3, 3 \mapsto 0\}\rangle_c$, ...

Ghost commands Custom chunks introduce a new problem. Some commands expect the presence of certain heap chunks on the heap. For example, $[x] := y$ with store $s_0[x := 2]$ requires a chunk $2 \mapsto v$ for some value v to execute successfully. However, it is possible this chunk is hidden inside a custom chunk. The verifier will not try to unfold these chunks on its own but instead relies on the user to open up these custom chunks himself using the **open** command. Conversely, it is possible to fold chunks back into one abstracting chunk using the **close** command. Thus, we need to extend SIL to accommodate these two commands. An important fact about these commands is that they have no effect on the semantics of a program; they merely aid the verifier to perform its job.

Symbols Despite the introduction of custom chunks, we are still plagued by execution trees with infinite breadth. For example, the **malloc** command generates an infinite number of states, as does routine call (more specifically, the production of assertions with unbound variables.) We make use of *symbols* [8] to take care of this issue.

A symbol represents an arbitrary value, meaning that a single symbolic state represents an infinite number of concrete states, one for each possible value assignment. However, this is too coarse an approximation: since a symbol can be replaced by *any* natural number, the information lost with respect to CE could make SE fail needlessly. For this, we add a third component to the symbolic state called the *path condition*, denoted Φ, allowing us to put restrictions on the range of values a symbol can represent. A symbolic state is denoted $\langle \Phi, \widehat{s}, \widehat{h}\rangle_s$, the hats being used to indicate their wearer's symbolic nature. For example, $\langle a < 2, \widehat{s}_0[x := b], \{b \mapsto a\}\rangle_s$ represents all concrete states where x points to some location containing either 0 or 1.

Nondeterminism One last complication arises from the possibility of ambiguous matches. For example, consider the heap $\{\mathrm{list}(a), \mathrm{list}(0)\}$. The command **open** $\mathrm{list}(\_)$ fails to indicate which of the list-chunks it refers to (the `open` command supports wildcards; the same is possible with ambiguous preconditions), so

[3] Concrete states are technically not allowed to contain custom chunks, but we allow it here for explanatory purposes.

that both possibilities need to be taken into account. This ambiguous matching again leads to nondeterminism in execution, but it is inherently of a different kind [9] than the one previously discussed. The nondeterminism expressed by symbols and custom chunks is *demonic*: execution must succeed for *every possible value* this symbol might represent and for *all possible unfoldings* of the chunk. This is comparable to a conjunction. The nondeterminism due to ambiguous matches however is *angelic*: success is only required for one of the possible matches and hence is rather similar to a disjunction.

Thus, we can distinguish two levels of nondeterminism: the angelic and the demonic level. To express these, we again generalize our execute function:

$$\text{execute} : \mathit{Statement} \longrightarrow \mathit{ProgramState} \longrightarrow \mathcal{P}(\mathcal{P}(\mathit{ProgramState}))$$

The resulting set-of-set-of-states must be interpreted as follows. Take for example $\{\{\sigma_1, \sigma_2\}, \{\sigma_3, \sigma_4, \sigma_5\}\}$. The "inner sets" correspond to the demonic level, while the "outer sets" represent the angelic level. More explicitly, if execution has to proceed, it has to succeed for either both σ_1 and σ_2, or for all three σ_3, σ_4 and σ_5. Put in logical terms, the set-of-sets-of-states can be seen as a disjunctive normal form: $(\sigma_1 \wedge \sigma_2) \vee (\sigma_3 \wedge \sigma_4 \wedge \sigma_5)$.

This representation allows us to represent failure elegantly, i.e. empty set. Even though the CE does not require angelic choice (as there are no ambiguous matches possible), we will also use this set-of-sets structure to formalise it for reasons of uniformity.

Soundness As mentioned earlier, the SE should be a finite approximation of the CE. The previous paragraphs dealt with the finite aspect; now we turn our attention to the approximation part. Our goal is to detect failures in the CE. The fact that there are infinitely many execution paths makes direct verification impossible. For SE to be a good approximation of the CE, or, in other words, for it to be sound, we expect it to fail if the corresponding CE fails. Turned around, we wish that successful SE implies successful CE.

3 Operators

In this section, we define the building blocks for the formal definitions. Section 3.1 defines what operators are, Sect. 3.2 shows how to combine them and Sect. 3.3 defines a few basic reusable operators. In order to be able to compare the output of operators, we also define containment in Sect. 3.4.

3.1 Definition

As explained previously, we will model the executions as functions returning sets-of-sets-of-states. Since commands often share functionality (e.g. many commands update the store), we will express the semantics of commands using composable *operators*. Readers may recognize monads [10, 11] and/or their implementation in Haskell.

Definition 1. *We define an* operator with state set S and result in A *as follows:*

$$\mathit{Operator}_A^S = S \to \mathcal{P}(\mathcal{P}(S \times A))$$

An operator thus takes an input state (e.g. a concrete state) and returns a number of output states, each accompanied by some result value. For example, an allocation operator could use this return value to indicate where the memory has been allocated on the heap.

Definition 2. *The empty set corresponds to* failure*. We introduce the abstracting notation R* ok *to mean $R \neq \emptyset$.*

3.2 Operator Binding

Operator binding allows us to build new operators from two smaller ones and can be interpreted as sequencing: each output state and result value pair from the left operator are fed to the right operator, after which the results are "reshuffled" in order to match the expected structure of set-of-sets-of-states.

Definition 3 (auxiliary definitions).

$$\prod_{i\in I} R(i) = \{f \mid f \in I \to \bigcup_{i\in I} R(i), \forall\, i \in I.\ f(i) \in R(i)\}$$

$$\bigodot_{i\in I} R(i) = \{ \bigcup_{i\in I} f(i) \mid f \in \prod_{i\in I} R(i) \}$$

$$\bigotimes_{i\in I} op(i) = \lambda\, \sigma.\ \bigodot_{i\in I} op(i)(\sigma)$$

$$op_1 \otimes op_2 = \bigotimes_{i\in\{1,2\}} op_i$$

Definition 4. *Operator binding combines an $\mathit{Operator}_A^S$ with a function $f : A \longrightarrow \mathit{Operator}_B^S$ to yield an $\mathit{Operator}_B^S$:*

$$op \oplus f = \lambda\, \sigma.\ \bigcup_{\Sigma'\in op(\sigma)} \bigodot_{(\sigma',r)\in\Sigma'} f(r)(\sigma')$$

These definitions deserve some extra explanation. We ignore the result values and focus solely on how the states are manipulated. We mentioned before how a set-of-sets-of-states can be interpreted as a logical formula in disjunctive normal form (DNF). Thus, given an initial state, the left operator yields such a disjunction of conjunctions of states. Applying the right operator amounts to applying it to every state in turn, each time yielding a new logical formula in DNF. This results in a disjunction of conjunctions of disjunctions of conjunctions, which we need to renormalize. The $\odot$ operation can be seen as taking the conjunction of DNF formulae and renormalizing them, while union takes their renormalized disjunction.

$$\vee\ \wedge \vee \wedge \quad \xrightarrow{\odot} \quad \vee\ \vee \wedge \quad \xrightarrow{\cup} \quad \vee \wedge$$

We also introduce the following shorthand notation:

$$op_1 \boxplus op_2 = op_1 \oplus \lambda\, r.\ op_2$$

In essence, this notation throws away the return value of the first operator, and will come in handy when dealing with operators of type $\mathit{Operator}^{S}_{\mathsf{unit}}$. The unit value is written $\square$.

3.3 Basic Operators

Figure 1 shows a few generic operators[4] which can be shared by all executions. The fail operator represents failure and removes an entire set-of-states (i.e. one option is removed on the angelic level.) stop allows us to short-circuit to success as it "eats up" anything that comes after it: $\mathsf{stop} \boxplus f = \mathsf{stop}$.

state and $\mathsf{set\text{-}state}$ are low-level operators allowing to manipulate the state directly. filter_a and filter_d filter on the angelic and demonic level, respectively. If a state does not satisfy the given predicate, the demonic filter will only remove that state, while the angelic filter removes the entire set-of-states to which it belongs. pick_a and pick_d are angelic and demonic choice, respectively: angelic picking requires execution to succeed for one of the elements in the given set, while demonic picking demands success for every element.

Definition 5. *We define the* do-notation *recursively as follows:*

$$\begin{array}{l}\mathbf{do}\ x \leftarrow op\\ \quad r\end{array} = op \oplus \lambda\, x.\ \mathbf{do}\ r \qquad \begin{array}{l}\mathbf{do}\ op\\ \quad r\end{array} = op \boxplus \mathbf{do}\ r \qquad \mathbf{do}\ op\ = op$$

We also use a single-line notation: $\mathbf{do}\ x \leftarrow op_1; y \leftarrow op_2; \ldots$

$$\begin{aligned}
\mathsf{return}(a) &= \lambda\, \sigma.\ \{\, \{\, (\sigma, a)\, \}\, \} & \mathsf{fail} &= \lambda\, \sigma.\ \emptyset\\
\mathsf{stop} &= \lambda\, \sigma.\ \{\, \emptyset\, \} & \mathsf{filter}_a(p) &= \lambda\, \sigma.\ \{\, \{\, (\sigma, \square)\, \} \mid p(\sigma)\, \}\\
\mathsf{nop} &= \mathbf{return}\ \square & \mathsf{filter}_d(p) &= \lambda\, \sigma.\ \{\, \{\, (\sigma, \square) \mid p(\sigma)\, \}\, \}\\
\mathsf{state} &= \lambda\, \sigma.\ \{\, \{\, (\sigma, \sigma)\, \}\, \} & \mathsf{pick}_a(A) &= \lambda\, \sigma.\ [\![\{\, \{\, (\sigma, x)\, \} \mid x \in A\, \}]\!]\\
\mathsf{set\text{-}state}(\sigma') &= \lambda\, \sigma.\ \{\, \{\, (\sigma', \square)\, \}\, \} & \mathsf{pick}_d(A) &= \lambda\, \sigma.\ \{\, \{\, (\sigma, x) \mid x \in A\, \}\, \}
\end{aligned}$$

Fig. 1. Basic Operators

3.4 Containment

Since we are dealing with two different executions, we need to be able to compare their results in some way. We ignore for a moment the fact that the executions manipulate different types of state and focus on how to compare two sets-of-sets.

[4] $[\![A]\!]$ takes the *union-closure* of A: it is the smallest superset of A for which $\forall\, X, Y \in A.\ X \cup Y \in A$. We sadly do not have enough room to delve into further details; suffice it to say it is necessary that operators yield union-closed results in order for the binding to be associative.

Definition 6. Containment *is a reflexive and transitive relation between sets of sets. We also allow it to be applied on operators.*

$$\begin{aligned} R_1 \Supset R_2 &\iff \forall\, \Sigma_1 \in R_1.\, \exists\, \Sigma_2 \in R_2.\, \Sigma_1 \supseteq \Sigma_2 \\ op_1 \Supset op_2 &\iff \forall\, \sigma.\, op_1(\sigma) \Supset op_2(\sigma) \end{aligned}$$

In order to make sense of this definition, we again switch to the logical view. The DNF formula expresses which states need to be taken into consideration when executing the next step: the more states, the heavier the burden. If SE were to yield fewer states than the CE, it would make it easier not to fail, compromising soundness. Ideally, the SE would produce the exact same states as the CE, achieving soundness *and* completeness, but in this paper we will settle for an overapproximation, sacrificing completeness.

Thus, the results from SE should "contain" the results of CE. If we were only dealing with the demonic level, it would be a mere matter of having the SE produce a superset of the CE, but the angelic level complicates this. In logical terms, we want the SE DNF formula to imply the CE one, or in other words, one disjunction must imply another one. For example,

$$\Sigma_1 \vee \Sigma_2 \;\Rightarrow\; \Sigma_1' \vee \Sigma_2' \vee \Sigma_3'$$

This is logically equivalent with

$$((\Sigma_1 \Rightarrow \Sigma_1') \vee (\Sigma_1 \Rightarrow \Sigma_2') \vee (\Sigma_1 \Rightarrow \Sigma_3')) \wedge ((\Sigma_2 \Rightarrow \Sigma_1') \vee (\Sigma_2 \Rightarrow \Sigma_2') \vee (\Sigma_2 \Rightarrow \Sigma_3'))$$

Set-wise, the implications corresponds to "being a superset of", leading to Def. 6.

Lemma 1. *Some key properties of the $\Supset$ relation are*

$$\begin{aligned} &op_1 \Supset op_2 \Rightarrow op_1 \text{ ok} \qquad \Rightarrow op_2 \text{ ok} \\ &op_1 \Supset op_2 \Rightarrow op_1' \Supset op_2' \Rightarrow op_1 \boxplus op_1' \Supset op_2 \boxplus op_2' \quad \text{all } op \in \mathit{Operator}^{S}_{\square} \end{aligned}$$

4 Execution Formalisations

We present formal definitions of what Sect. 2 explained informally.

4.1 Small Imperative Language

We define the syntax of the small imperative language (SIL). As explained in Sect. 2.3, the SE needs extra help in the form of routine specification and chunk manipulation commands. For this, we need to extend SIL, giving us SIL^{++}. These additions are marked with $\langle \ldots \rangle_{+}$.

Definition 7 (SIL/SIL^{++}).

$$
\begin{aligned}
e &::= n \mid x \mid e +_{\mathrm{e}} e \mid e -_{\mathrm{e}} e \\
b &::= e =_{\mathrm{b}} e \mid e <_{\mathrm{b}} e \mid \neg_{\mathrm{b}} b \\
c &::= \textbf{\textit{skip}} \mid x := e \mid x := [x] \mid [x] := x \mid c; c \mid r(x) \\
&\quad \mid \textbf{if } b \textbf{ then } c \textbf{ else } c \mid x := \textbf{\textit{malloc}}(n) \mid \textbf{\textit{free}}(x) \\
&\quad \mid \langle \textbf{\textit{open}}\, p(e, \_) \rangle_{+} \mid \langle \textbf{\textit{close}}\, p(e, e) \rangle_{+} \\
a &::= b \mid p(e, ?x) \mid a \star a \mid \textbf{if } b \textbf{ then } a \textbf{ else } a \\
\mathit{Routine} &::= \textbf{\textit{routine}}\, r(x)\ \langle \textbf{\textit{req}}\, a\ \textbf{\textit{ens}}\, a \rangle_{+} = c \\
\mathit{UserPredicate} &::= \textbf{\textit{predicate}}\, p(x, y) = a \\
\mathit{Program} &\ = \mathcal{P}(\mathit{Routine})\ \langle \times \mathcal{P}(\mathit{UserPredicate}) \rangle_{+}
\end{aligned}
$$

Definition 8. *The translation translate*(c) *of a SIL^{++} command c into a SIL command is accomplished by removing the routine specification and replacing* ***open*** *and* ***close*** *by* ***skip****.*

4.2 Concrete Execution

The CE represents the semantics of a SIL command and mimics how a program would run on a machine with unlimited memory. The program state consists of a store (a function mapping variable names to values) and a heap, represented by a multiset of chunks. This deviates from the regular heap representation in other formalisations (i.e. a function mapping memory locations to values); the reason for this is to simplify making the jump towards symbolic execution i.e. it eases the addition of custom chunks.

Definition 9 (concrete execution).

$$
\begin{aligned}
s \in \quad & \mathit{CStore} = \mathit{Symbol} \to \mathbb{N} \\
p \in \quad & \mathit{CPredicate} = \{\mapsto, \mathsf{mb}\} \\
\alpha \in \quad & \mathit{CChunk} = \{p(\ell, v) \mid p \in \mathit{CPredicate},\ \ell, v \in \mathbb{N}\} \\
h \in \quad & \mathit{CHeap} = \mathit{CChunk} \to \mathbb{N} \\
\sigma = \langle s, h \rangle_c \in \quad & \mathit{CState} = \mathit{CStore} \times \mathit{CHeap} \\
& s_0 = \lambda\, x.\, 0
\end{aligned}
$$

Space limitations have forced us to leave out the definition of the concrete execution. We will point out however that the execution is nondeterministic, due to **malloc**'s semantics: the $\mathsf{alloc}(n)$ operator will demonically pick a memory address and add a $\mathsf{mb}(\ell, n)$ and $\ell + i \mapsto v_i$ chunks for $i = 0 \ldots n - 1$ and $v_i \in \mathbb{N}$ to the heap.

Our goal is to detect potential failures. There are three ways to fail in CE: reading and writing from the heap at location ℓ expect a chunk $\ell \mapsto v$ for some v to be present on the heap, otherwise failure ensues. This is similar to dereferencing wild or dangling pointers in C. Freeing memory at location ℓ requires a $\mathsf{mb}(\ell, n)$ chunk accompanied by $\ell + i \mapsto v_i$ for $i = 0 \ldots n$. Their absence corresponds to trying to free unallocated memory.

4.3 Symbolic Execution

The SE is a finite version of the CE. Where the CE produces an infinite number of states, the SE relies on symbolic states (making use of symbols and heap abstraction) to avoid this issue: a single symbolic state represents an infinite number of concrete states, which makes it possible to represent the infinite output of CE by a finite number of symbolic states.

Definition 10. *The* symbolic execution *is defined in Fig. 4, with Fig. 2 and Fig. 3 containing the definitions of the auxiliary operators.*

$$\begin{array}{rrl}
n \in \mathbb{N} & \varsigma \in \mathit{Symbol} & \widehat{s}_0 = \lambda\, x.\, 0 \\
t \in & \mathit{Term} & ::= n \mid \varsigma \mid t +_{\mathrm{t}} t \mid t -_{\mathrm{t}} t \\
\phi \in & \mathit{Formula} & ::= t =_{\mathrm{f}} t \mid t <_{\mathrm{f}} t \mid \neg_{\mathrm{f}}\, \phi \\
p \in & \mathit{SPred} & = \{\mathsf{mb}, \mapsto\} \cup \mathit{UserDefinedPreds} \\
\widehat{\alpha} \in & \mathit{SChunk} & = \{p(t, \widehat{t'}) \mid p \in \mathit{SPred}, t, t' \in \mathit{Term}\} \\
\Phi \in & \mathit{PathCondition} & = \mathcal{P}(\mathit{Formula}) \\
\widehat{s} \in & \mathit{SStore} & = \mathcal{X} \to \mathit{Term} \\
\widehat{h} \in & \mathit{SHeap} & = \mathit{SChunk} \to \mathbb{N} \\
\widehat{\sigma} = \langle \Phi, \widehat{s}, \widehat{h} \rangle_s \in & \mathit{SState} & = \mathit{PathCondition} \times \mathit{SStore} \times \mathit{SHeap}
\end{array}$$

The attentive reader might wonder about the reason why fresh-symbol adds the seemingly useless $\varsigma = \varsigma$ to the path condition. Without it, two consecutive calls to fresh-symbol could yield the same symbol. We also left out some function and operator definitions (e.g. to − term, to-formula, eval) which are both too trivial and too lengthy to justify their inclusion in the paper.

4.4 State Concretisation

Each symbolic state represents a potentially infinite amount of concrete states. We perform this concretisation in two steps: we first replace symbols by values and then unfold all custom chunks. The in-between state is called a semi-concrete state. We have also defined a corresponding semi-concrete execution (see the full report [7]), but we don't discuss it in detail in this paper. Note that the set of concrete states is a subset of the set of semi-concrete states.

Definition 11. *The $\mathcal{R}$ interpretation function applies all possible symbol/value substitutions which satisfy the path condition.*

$$\mathcal{R}(\langle \Phi, \widehat{s}, \widehat{h} \rangle_s) = \{\ \langle [\widehat{s}]_I, [\widehat{h}]_I \rangle_{sc} \mid I \vDash \Phi\ \}$$

The ρ operator demonically replaces the current state by an interpretation of it.

$$\rho = \mathbf{do}\ \sigma \leftarrow \mathit{state}; \sigma' \leftarrow \mathit{pick}_d(\mathcal{R}(\sigma)); \mathit{set\text{-}state}(\sigma')$$

Definition 12 (entailment).

$$\langle s, h \rangle_{sc} \vDash a \quad \Leftrightarrow \quad \begin{pmatrix} \mathbf{do}\ \mathit{consume}_a(a) \\ \mathit{leak\text{-}check} \end{pmatrix} (\langle s, h \rangle_{sc}) \neq \emptyset$$

```
store = do $\langle \Phi, \widehat{s}, \widehat{h} \rangle_s$ ← state
           return $\widehat{s}$
set-store($\widehat{s}'$) = do $\langle \Phi, \widehat{s}, \widehat{h} \rangle_s$ ← state
           set-state($\langle \Phi, \widehat{s}', \widehat{h} \rangle_s$)
r-store($x$) = do $\widehat{s}$ ← store
           return $\widehat{s}(x)$
w-store($x, t$) = do $\widehat{s}$ ← store
           set-store($s[x := t]$)
havoc($x$) = do $\widehat{v}$ ← fresh-symbol
           w-store($x, \widehat{v}$)
local($\widehat{s}, op$) = do $\widehat{s}'$ ← store
           set-store($\widehat{s}$)
           $r \leftarrow op$
           set-store($\widehat{s}'$)
           return $r$
assert($b$) = do $\phi$ ← to-formula($b$)
           smt-check($\phi$)
assume($b$) = do $\phi$ ← to-formula($b$)
           add-formula($\phi$)
```

```
heap, set-heap = ...
pathcond, set-pathcond = ...

leak-check = filter$_a$($\lambda\ \langle \Phi, \widehat{s}, \widehat{h} \rangle_s .\ h = \emptyset$)
eval($e$) = do $\widehat{s}$ ← store
           return $\mathrm{to\text{-}term}(s, e)$
alloc($n$) = do $\widehat{\ell}$ ← fresh-symbol
           alloc′($\widehat{\ell}, n$)
           produce$_c$(mb($\widehat{\ell}, n$))
           return $\widehat{\ell}$
alloc′($\widehat{\ell}, 0$) = return $\square$
alloc′($\widehat{\ell}, n+1$) = do $\widehat{v}$ ← fresh-symbol
           produce$_c$($\widehat{\ell} +_{\mathrm{t}} n \mapsto \widehat{v}$)
           alloc′($\widehat{\ell}, n$)
```

```
w-heap($\widehat{\ell}, \widehat{v}'$) = do ($\widehat{\ell}' \mapsto \widehat{v}$) ← find-chunk($\widehat{\ell} \mapsto \_$)
           consume$_c$($\widehat{\ell}' \mapsto \widehat{v}$)
           produce$_c$($\widehat{\ell}' \mapsto \widehat{v}'$)
r-heap($\widehat{\ell}$) = do ($\widehat{\ell}' \mapsto \widehat{v}$) ← find-chunk($\widehat{\ell} \mapsto \_$)
           return $\widehat{v}$
produce$_c$($p(\widehat{\ell}, \widehat{v})$) = do $\widehat{h}$ ← heap
           set-heap($\widehat{h} \uplus \{p(\widehat{\ell}, \widehat{v})\}$)
consume$_c$($p(\widehat{\ell}, \widehat{v})$) = do $\widehat{h}$ ← heap
           set-heap($\widehat{h} - \{p(\widehat{\ell}, \widehat{v})\}$)
fresh-symbol = $\lambda\ \langle \Phi, \widehat{s}, \widehat{h} \rangle_s .\ \Big\{ \{ (\langle \Phi \cup \{\varsigma =_{\mathrm{f}} \varsigma\}, \widehat{s}, \widehat{h} \rangle_s, \varsigma) \} \Big\}$
           for some $\varsigma$ not appearing in $\Phi$
smt-check($\phi$) = filter$_a$($\lambda\ \langle \Phi, \widehat{s}, \widehat{h} \rangle_s .\ \Phi \vdash_{\mathrm{SMT}} \phi$)
find-chunk($p(\widehat{\ell}, \_)$) = do $\widehat{h}$ ← heap
           $p(\widehat{\ell}', \widehat{v}')$ ← pick$_a$($\{p(\widehat{\ell}', \widehat{v}') \mid \widehat{v}' \in \mathcal{S}, p(\widehat{\ell}', \widehat{v}') \in \widehat{h}\}$)
           smt-check($\widehat{\ell} =_{\mathrm{f}} \widehat{\ell}'$)
           return $p(\widehat{\ell}', \widehat{v}')$
add-formula($\phi$) = do filter$_d$($\lambda\ \langle \Phi, \widehat{s}, \widehat{h} \rangle_s .\ \Phi \not\vdash_{\mathrm{SMT}} \neg\phi$)
           $\Phi$ ← pathcond
           set-pathcond($\Phi \cup \{\phi\}$)
consume-cells($\widehat{\ell}, 0$) = return $\square$
consume-cells($\widehat{\ell}, n+1$) = do $\widehat{h}$ ← heap
           ($\widehat{\ell}' \mapsto \widehat{v}$) ← find-chunk($\widehat{\ell} +_{\mathrm{t}} n \mapsto \_$)
           consume$_c$($\widehat{\ell}' \mapsto \widehat{v}$)
           consume-cells($\widehat{\ell}, n$)
```

Fig. 2. Symbolic Execution: Auxiliary Operators

$$
\begin{aligned}
\mathsf{produce}_a(b) &= \mathsf{assume}(b)\\
\mathsf{produce}_a(a * a') &= \mathsf{produce}_a(a) \boxplus \mathsf{produce}_a(a')\\
\mathsf{produce}_a(\textbf{if } b \textbf{ then } a \textbf{ else } a') &= \begin{pmatrix}\textbf{do } \mathsf{assume}(b)\\ \quad \mathsf{produce}_a(a)\end{pmatrix} \otimes \begin{pmatrix}\textbf{do } \mathsf{assume}(\neg b)\\ \quad \mathsf{produce}_a(a')\end{pmatrix}\\
\mathsf{produce}_a(p(e, ?x)) &= \textbf{do } \ell \leftarrow \mathsf{eval}(e)\\
&\qquad \widehat{v} \leftarrow \mathsf{fresh\text{-}symbol}\\
&\qquad \mathsf{produce}_c(p(\widehat{\ell}, \widehat{v}))\\
&\qquad \mathsf{w\text{-}store}(x, \widehat{v})
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{consume}_a(b) &= \mathsf{assert}(b)\\
\mathsf{consume}_a(a * a') &= \mathsf{consume}_a(a) \boxplus \mathsf{consume}_a(a')\\
\mathsf{consume}_a(\textbf{if } b \textbf{ then } a \textbf{ else } a') &= \begin{pmatrix}\textbf{do } \mathsf{assume}(b)\\ \quad \mathsf{consume}_a(a)\end{pmatrix} \otimes \begin{pmatrix}\textbf{do } \mathsf{assume}(\neg b)\\ \quad \mathsf{consume}_a(a')\end{pmatrix}\\
\mathsf{consume}_a(p(e, ?y)) &= \textbf{do } \widehat{\ell} \leftarrow \mathsf{eval}(e)\\
&\qquad p(\widehat{\ell}', \widehat{v}') \leftarrow \mathsf{find\text{-}chunk}(p(\widehat{\ell}, \_))\\
&\qquad \mathsf{consume}_c(p(\widehat{\ell}', \widehat{v}'))\\
&\qquad \mathsf{w\text{-}store}(y, \widehat{v}')
\end{aligned}
$$

Fig. 3. Assertion production and consumption

```
s-exec(skip) =
  return $\square$
s-exec($x := e$) =
  do $\widehat{v} \leftarrow$ eval($e$)
     w-store($x, \widehat{v}$)
s-exec($c; c'$) =
  do s-exec($c$)
     s-exec($c'$)
s-exec($x :=$ malloc($n$)) =
  do $\widehat{\ell} \leftarrow$ alloc($n$)
     w-store($x, \widehat{\ell}$)
s-exec(free($x$)) =
  do $\widehat{\ell} \leftarrow$ r-store($x$)
     $n \leftarrow$ block-size($\widehat{\ell}$)
     consume-cells($\widehat{\ell}, n$)
     consume$_c$(mb($\widehat{\ell}, n$))
s-exec($x := [x']$) =
  do $\widehat{\ell} \leftarrow$ r-store($x'$)
     $\widehat{v} \leftarrow$ r-heap($\widehat{\ell}$)
     w-store($x, \widehat{v}$)
```

```
s-exec($[x] := x'$) =
  do $\widehat{\ell} \leftarrow$ r-store($x$)
     $\widehat{v} \leftarrow$ r-store($x'$)
     w-heap($\widehat{\ell}, \widehat{v}$)
s-exec(open $p(e, \_)$) =
  do $\widehat{\ell} \leftarrow$ eval($e$)
     $p(\widehat{\ell}', \widehat{v}') \leftarrow$ find-chunk($p(\widehat{\ell}, \_)$)
     consume$_c$($p(\widehat{\ell}', \widehat{v}')$)
     local($\widehat{s}_0[x := \widehat{\ell}'][y := \widehat{v}']$, produce$_a$($a$))
  where predicate $p(x, y) = a$
s-exec(close $p(e, e')$) =
do $t \leftarrow$ eval($e$)
   $t' \leftarrow$ eval($e'$)
   local($\widehat{s}_0[x := t][y := t']$, consume$_a$($a$))
   produce$_c$($p(t, t')$)
  where predicate $p(x, y) = a$
s-exec($r(x)$) =
  do $\widehat{v} \leftarrow$ r-store($x$)
     local($\widehat{s}_0[x := v]$, consume$_a$($a$) $\boxplus$ produce$_a$($a'$))
  with routine $r(x)$ requires $a$ ensures $a'$
```

$$
\mathsf{s\text{-}exec}(\textbf{if } b \textbf{ then } c \textbf{ else } c') = \begin{pmatrix}\textbf{do } \mathsf{assume}(b)\\ \quad \mathsf{s\text{-}exec}(c)\end{pmatrix} \otimes \begin{pmatrix}\textbf{do } \mathsf{assume}(\neg b)\\ \quad \mathsf{s\text{-}exec}(c')\end{pmatrix}
$$

Fig. 4. Symbolic execution

Definition 13. *The* heap refinement *relation* $\trianglelefteq$ *between a concrete heap* h_c *and a semi-concrete heap* h_{sc} *indicates* h_c *is a fully unfolded version of* h_{sc}.

$$\frac{}{h \trianglelefteq h}\ \trianglelefteq\text{-REFLEXIVITY} \qquad \frac{h_c \trianglelefteq h_{sc} \qquad h'_c \trianglelefteq h'_{sc}}{h_c \uplus h'_c \trianglelefteq h_{sc} \uplus h'_{sc}}\ \trianglelefteq\text{-UNION}$$

$$\frac{h_c \trianglelefteq h_{sc} \qquad \textbf{predicate}\ p(x, x') = a \qquad \langle s_0[x := \ell][x' := v], h_{sc}\rangle_{sc} \vDash a}{h_c \trianglelefteq \{\ p(\ell, v)\ \}}\ \trianglelefteq\text{-PREDICATE}$$

Definition 14. *The function* $\mathcal{K}$ *unfolds a heap of a given semi-concrete state, yielding a set of concrete states called the* refinement set.

$$\mathcal{K}(\langle s, h_{sc}\rangle_{sc}) = \{\langle s, h_c\rangle_c \mid h_c \trianglelefteq h_{sc}\}$$

The κ *operator demonically replaces the current state by one of its unfoldings.*

$$\kappa = \textbf{do}\ \sigma \leftarrow \mathsf{state}; \sigma' \leftarrow \mathsf{pick}_d(\mathcal{K}(\sigma)); \mathsf{set\text{-}state}(\sigma')$$

5 Soundness

Our goal is to detect failing concrete executions through the use of the SE. We achieve this by pulling the program apart and symbolically executing each routine in turn, making sure that given a state which satisfies the precondition, execution of the routine's body leads to an end state where the precondition holds (see Def. 15).

Definition 15. *Given a routine* r *defined as* $\textbf{routine}\ r(x)\ \textbf{req}\ a\ \textbf{ens}\ a' = c$, *we say* r *is* valid *if*

$$\mathit{valid\text{-}routine}(r) = \begin{pmatrix} \textbf{do}\ \mathsf{havoc}(x) \\ \quad s \leftarrow \mathsf{store} \\ \quad \mathsf{produce}_a(a) \\ \quad \mathsf{local}(s, \mathsf{execute}(c)) \\ \quad \mathsf{consume}_a(a') \\ \quad \mathsf{leak\text{-}check} \end{pmatrix} (\langle \emptyset, \widehat{s}_0, \emptyset\rangle_s) \neq \emptyset$$

Definition 16. *A program is* valid *if each of its routines is valid.*

Definition 17. *The symbolic execution is* sound *if, given a valid program and assuming c is the body of the main routine,*

$$\mathsf{s\text{-}exec}(c)(\sigma_s)\ \text{ok} \quad \Rightarrow \quad \mathsf{c\text{-}exec}(\mathit{translate}(c))(\sigma_c)\ \text{ok}$$

for any concretisation σ_c *of* σ_s.

Separation logic [5] defines separating conjunction, which expresses that its left and right operand are true in disjunct parts of the heap. This allows for a very elegant frame rule: $\{P\}c\{Q\} \;\Rightarrow\; \{P \star R\}c\{Q \star R\}$. In other words, if a command c transforms a state satisfying P into a state satisfying Q, we can add arbitrary chunks R to the heap, and they will be preserved. We can express this property as follows:

Definition 18. *An operator op is* production-agnostic *iff*

$$op \oplus (\lambda\, r.\ \mathit{produce}_c(\alpha) \boxplus \mathit{return}\ r) \ \Supset\ \mathit{produce}_c(\alpha) \boxplus op$$

Lemma 2. $\mathit{s\text{-}exec}(c) \boxplus \rho \boxplus \kappa \ \Supset\ \rho \boxplus \kappa \boxplus \mathit{c\text{-}exec}(c') \qquad$ *with* $c' = \mathit{translate}(c)$

Proof. We limit ourselves to a proof sketch. For simplicity, we defined the semi-concrete execution, which has heap abstraction but makes no use of symbols. The proof goal then becomes

$$\mathsf{sc\text{-}execute}(c) \boxplus \kappa \Supset \kappa \boxplus \mathsf{c\text{-}exec}(c') \qquad \mathsf{s\text{-}exec}(c) \boxplus \rho \Supset \rho \boxplus \mathsf{sc\text{-}execute}(c)$$

Both parts are proved by structural induction on c. One interesting aspect is how to deal with the routine call while dealing with the left subgoal: for this, we need to show that the result of the consumption-production (Fig. 4) "contains" the same states as normally executing the routine's body, after which the induction hypothesis takes care of the rest. From routine validity follows

$$\mathsf{nop} \Supset \mathsf{local}(s_0[x := v], \mathbf{do}\ \mathsf{produce}_a(a); \mathsf{local}(s_0[x := v], \mathsf{sc\text{-}exec}(c)); \mathsf{consume}_a(a'))$$

This allows us to insert the rhs in the definition (Fig. 4) of $\mathsf{s\text{-}exec}(r(x))$:

$$\begin{aligned}\mathbf{do}\ & \widehat{v} \leftarrow \mathsf{r\text{-}store}(x); s \leftarrow \mathsf{store}; \mathsf{set\text{-}store}(\widehat{s}_0[x := v]); \mathsf{consume}_a(a);\\ & \mathsf{local}(s_0[x := v], \mathbf{do}\ \mathsf{produce}_a(a); \mathsf{local}(s_0[x := v], \mathsf{sc\text{-}exec}(c)); \mathsf{consume}_a(a'));\\ & \mathsf{produce}_a(a'); \mathsf{set\text{-}store}(s)\end{aligned}$$

Simplifying this yields the desired result.

Theorem 1. *The symbolic execution is a sound.*

Proof. Follows from Lemma 2.

6 Related Work

Smallfoot [12] is another verification tool based on separation logic [5] for which there exists a formalisation and soundness proof [13]. jStar [14] is also based on separation logic and uses CoreStar [15] as its foundation.

Another approach to verification is based on predicate transformers, e.g. Dijkstra's weakest preconditions [16]. Many verifiers are built on top of this framework. For example, VCC [17], Dafny [18], Chalice [19] and Spec♯ [20] are based on Boogie ([21, 22]), which defines an intermediate language similar to ours, but is used instead to generate a verification condition, whose validity implies the correctness of the program. A machine-checked soundness proof for the weakest precondition technique is available [23, 24].

References

1. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the VeriFast program verifier. In: Programming Languages and Systems (APLAS). (2010)
2. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: Verifast: A powerful, sound, predictable, fast verifier for C and Java. In: NFM. (2011)
3. Philippaerts, P., Vogels, F., Smans, J., Jacobs, B., Piessens, F.: The Belgian electronic identity card: a verification case study. In: AVOCS. (2011)
4. Jacobs, B., Smans, J., Piessens, F.: Verification of unloadable modules. In: 17th International Symposium on Formal Methods. (2011)
5. Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. In: LICS. (2002)
6. Vogels, F., Jacobs, B., Piessens, F., Smans, J.: Annotation inference for separation logic based verifiers. In: Formal Techniques for Distributed Systems. (2011)
7. Vogels, F., Jacobs, B., Piessens, F.: Featherweight VeriFast: Extended Version. Technical Report CW614, KULeuven (2012)
8. King, J.C.: Symbolic execution and program testing. Commun. ACM **19** (July 1976)
9. Cavalcanti, A., Woodcock, J., Dunne, S.: Angelic nondeterminism in the unifying theories of programming. Formal Aspects of Computing **18**(3) (September 2006)
10. Wadler, P.: Monads for functional programming. In: Advanced Functional Programming. Lecture Notes in Computer Science. (1995)
11. Moggi, E.: An abstract view of programming languages. Technical report, Edinburgh University (1989)
12. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: FMCO. (2005)
13. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic Execution with Separation Logic. In: APLAS. (2005)
14. Distefano, D., Parkinson, M.J.: jStar: towards practical verification for Java. In: OOPSLA. (2008)
15. Botinčan, M., Distefano, D., Dodds, M., Griore, R., Naudžiūnienė, Parkinson, M.J.: coreStar: The core of jstar. In: Boogie. (2011)
16. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
17. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: TPHOLs. (2009) 23–42
18. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: LPAR-16. (2010)
19. Leino, K.R.M., Müller, P.: A basis for verifying multi-threaded programs. In: ESOP. (2009)
20. Barnett, M., Leino, Schulte, W. In: The Spec# Programming System: An Overview. (2005)
21. Barnett, M., Chang, B.E., Deline, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO. (2006)
22. Leino, K., Rümmer, P.: A Polymorphic Intermediate Verification Language: Design and Logical Encoding. In: TACAS. (2010)
23. Vogels, F., Jacobs, B., Piessens, F.: A machine checked soundness proof for an intermediate verification language. In: LNCS. (2009)
24. Vogels, F., Jacobs, B., Piessens, F.: A machine-checked soundness proof for an efficient verification condition generator. In: SAC. (2010)