Novel Designs for Memory Checkers Using Semantics and Digital Sequential Circuits

Mohamed A. El-Zawawy[1,2](✉)

[1] College of Computer and Information Sciences,
Al Imam Mohammad Ibn Saud Islamic University (IMSIU),
Riyadh, Kingdom of Saudi Arabia
[2] Department of Mathematics, Faculty of Science,
Cairo University, Giza 12613, Egypt
maelzawawy@cu.edu.eg

Abstract. Memory safety breaches have been main tools in many of the latest security vulnerabilities. Therefore memory safety is critical and attractive property for any piece of code. Separation logic can be realized as a mathematical tool to reason about memory safety of programs. An important technique for modern parallel programming is multithreading. For a multi-threaded model of programming (*Core-Par-C*), this paper introduces an accurate semantics which is employed to mathematically prove the undecidability of memory-safety of *Core-Par-C* programs. The paper also proposes a design for a hardware to act as an efficient memory checker against memory errors.

Keywords: Operational semantics · Separation logic · Memory-safety · Parallel programs · Digital sequential circuits

1 Introduction

Memory safety breaches were used extensively in many of the latest security vulnerabilities [10,19,31]. This reflects how critical and attractive the property of memory safety is for any piece of code. Therefor not only does the absence of memory safety result in software defect which in turn results in abnormal termination of program executions, but also this absence can be employed maliciously towards security vulnerabilities. Memory safety takes several forms including memory leaks, dangling pointers, and buffer overflows [35].

In presence of shared mutable data structure and to reason about imperative programs, separation logic [26] was designed as enrichment of Hoare logic. Therefore separation logic may be defined as a mathematical tool to reason about memory safety of imperative programs. The enrichment included extending the assertion language with a "separating conjunction" to express that several sub-assertions hold for different regions of the memory. Also a "separating implication" was added to the assertion language of Hoare logic. Defining assertions inductively and the new assertions resulted in flexible and precise depiction of memories with regulated sharing [16].

An important technique for modern programming is multithreading [34]. The use of multiply threads is useful in many direction including (a) building interactive servers that are capable of connecting with multiple clients in parallel, (b)utilizing parallelism of multiprocessors that share memory, and (c) building complex user interfaces. Hence studying multithreaded programs and their memory safety are crucial and attract growing interest.

For a multi-threaded model of programming (*Core-Par-C*), this paper presents a formal semantics that is used to mathematically prove the undecidability of memory-safety of *Core-Par-C* programs. The paper also illustrates special cases when the memory-safety problem become decidable.

Shared mutable data structures are used by many areas like artificial intelligence and systems programming. These structures are typically mutable because there are many points for updating and referencing the fields of the data structures. Techniques for reasoning about this approach have been researched for many decades. Either extremely complex or not applicable (even to code of moderate length) techniques are mostly currently used to carry out this reasoning process. Very little research were done to achieve such reasoning process using hardware. However such hardware seems like the convenient solution for the complexity and scalability issues.

This paper also presents a design for a hardware to act as a memory checker against memory errors. The hardware is a digital sequential circuit. Basic operations used in designing the hardware are those used in presenting the separation logic. Also memory states modeled in the hardware design are those considered in separation logic. Therefore this hardware may be realized as a main step towards designing a digital sequential circuit to carry the verifications of separation logic.

More preciously, the second contribution of this paper can be realized as a first attempt to achieve the separation logic as a reasoning tool for shared mutable data structures using digital sequential circuites. Four type of commands are basics for the separation logic: allocation, disposal, mutation, and looking up. Therefore our proposed technique establish codes to these operations (Figure 5). There are four states of memory (an empty heap, a singleton heap, a separation heap, and an error) in separation logic to reason about the memory. The error memory is a memory being treated illegally (may be under attack) and the singleton memory has a single allocated cell. The state where many septated cells are allocated in the memory is denoted by the terminology *separation heap*. The four types of commands and that of memory states are the basics of the design of the proposed digital sequential circuite.

Contributions

This paper has the following contributions:

1. A formal proof that memory safety of multithreaded programs is undecidable.
2. A design for a hardware to act as a memory checker against memory errors.

Paper Outline

The rest of the paper is organized as follows. Section 2 presents the first contribution of this paper; proving the the undecidability of memory safety of multithreaded programs. Section 3 presents the second contribution of this paper; a hardware to carry memory checks. Related and future work are discussed in Section 4. The paper is concluded in Section 5.

2 Memory Checker Using Dynamic Semantics

For a multi-threaded proposed model of programming (*Core-Par-C*), this section presents a formal semantics which is later used to discuss the decidability of memory-safety of *Core-Par-C*. The section also proposes solutions to over come memory-safety difficulties discussed in the section as well. To do so, the memory-safety in *Core-Par-C* is defined formally to express that the safety of a program amounts to being safe through any potential execution of *Core-Par-C*. This also amounts to being safe under any potential effects that the semantics of the parallel command (*par*) and the memory allocation statement (*malloc*) may have. Generally and practically, the concept of memory safety, is undecidable due to the undecidability nature of termination.

For terminating *Core-Par-C* programs, this section proves undecidability of memory-safety. The consequences of this is a believe that any semantics of *Core-Par-C* is not able of statically or dynamically detecting memory-safety problems even for terminating programs.

$$\begin{aligned}
& n \in \text{Integers},\ v \in \textit{Variables}\ ,\ \textit{and} \otimes \in \{-,+,\times\}\\
a \in \textit{A-expressions} &::= v \mid n \mid a_1 \otimes a_2\\
o \in \textit{B-expressions} &::= 1 \mid 0 \mid \neg o \mid a_1 \leq a_2 \mid a_1 = a_2 \mid o_1 \wedge o_2 \mid o_1 \vee o_2\\
c \in \textit{commands} &::= *v := a \mid v := a \mid \textit{malloc}(n) \mid v_1 := *v_2 \mid \textit{free}(n) \mid c_1; c_2 \mid\\
&\quad \textit{if } o \textit{ then } c_t \textit{ else } c_f \mid \textit{while } o \textit{ do } c_t \mid \textit{par}\{\{c_1\},\ldots,\{c_n\}\} \mid\\
&\quad \textit{par-if}\{(o_1,c_1),\ldots,(o_n,c_n)\} \mid \textit{par-for}\{c\}.
\end{aligned}$$

Fig. 1. *Core-Par-C*: A Programming Model for Multithreaded Programming with Pointers

Figure 1 presents the syntax of our model for multithreaded programming with pointers; *Core-Par-C*. *Variables* is a finite set of program variables. There are three main commands to express the multi-threaded nature of programming. These commands are *par*$\{\{c_1\},\ldots,\{c_n\}\}$ for parallel execution of commands, *par-if*$\{(o_1,c_1),\ldots,(o_n,c_n)\}$ for conditionally parallel execution of commands, and *par-for*$\{c\}$ for executing a randomly-chosen number of copies of c in parallel.

The following definition (Definition 1) presents the states of our proposed semantics.

Definition 1. *1. $E \in Env = Var \rightarrow Integers$.*
2. $M \in Mem = Integers^{+} \rightharpoonup Integers$.
3. $P \in Ptr = Integers^{+} \rightarrow Integers^{+}$.
4. A state is either an abort or a triple (E, M, P).

Boolean and arithmetic expressions semantics are built as usual. However we dot no allow pointers to get involved in Boolean and arithmetic operations. This is given in Figure 2. Semantics of statements of *Core-Par-C* is given in Figure 3.

$$\lessdot n \gtrdot E = n \quad \lessdot v \gtrdot E = E(v) \quad \lessdot \mathit{1} \gtrdot E = \mathit{true} \quad \lessdot \mathit{0} \gtrdot E = \mathit{false}$$

$$\lessdot a_1 \otimes a_2 \gtrdot E = \lessdot a_1 \gtrdot E \otimes \lessdot a_2 \gtrdot E$$

$$\lessdot \neg o \gtrdot E = \mathit{if}\ \lessdot o \gtrdot E\ \mathit{then\ false\ else\ true}$$

$$\lessdot a_1 = a_2 \gtrdot E = \mathit{if}\ (\lessdot a_1 \gtrdot E = \lessdot a_2 \gtrdot E)\ \mathit{then\ true\ else\ false}$$

$$\lessdot a_1 \leq a_2 \gtrdot E = \mathit{if}\ (\lessdot a_1 \gtrdot E \leq \lessdot a_2 \gtrdot E)\ \mathit{then\ true\ else\ false}$$

$$\lessdot o_1 \wedge o_2 \gtrdot (E, M, P) = \mathit{if}\ (\lessdot o_1 \gtrdot (E, M, P))\ \mathit{then}\ \lessdot o_2 \gtrdot (E, M, P)\ \text{else false}$$

$$\lessdot o_1 \vee o_2 \gtrdot E = \mathit{if}\ (\lessdot o_1 \gtrdot E)\ \mathit{then\ true\ else}\ \lessdot o_2 \gtrdot E$$

Fig. 2. Semantics of Boolean and Arithmetic Expressions of *Core-Par-C*

Definition 2 introduces the formal definition of memory safety of *Core-Par-C* programs.

Definition 2. *A program in Core-Par-C is terminating if it has an execution path in our proposed semantics (Figures 2 and 3) that does not lead to an abort. A program in Core-Par-C is memory-safe if for all its possible execution pathes it is terminating.*

Definition 3 presents two programs that paly a vital rule in proving the undecidability of the memory safety of *Core-Par-C* programs.

Definition 3. *We let unsafe and safe be the Core-Par-C programs defined as follows:*

– $\mathit{unsafe} \equiv \mathit{par}\{x := \mathit{malloc}(1), y := *x\}$, *and*
– $\mathit{safe} \equiv x := \mathit{malloc}(1); \mathit{par}\{z := *x, y := *x\}$.

Theorem 1[1] uses Definitions 2 and 3 to introduce and formally prove the undecidability of memory safety of terminating and non-terminating *Core-Par-C* programs.

[1] This theorem can be realized as a generalization of the work in [28].

$$\frac{}{v := a : (E, M, P) \twoheadrightarrow (E[v \mapsto \lessdot a \gtrdot E], M, P)}$$

$$\frac{E(v) \in dom(M)}{*v := a : (E, M, P) \twoheadrightarrow (E, M[E(v) \mapsto \lessdot a \gtrdot E], P)}$$

$$\frac{E(v) \notin dom(M)}{*v := a : (E, M, P) \twoheadrightarrow abort} \qquad \frac{E(v_2) \in dom(M)}{v_1 := *v_2 : (E, M, P) \twoheadrightarrow (E[v_1 \mapsto M(E(v_2))], M, P)}$$

$$\frac{M' = M \otimes M'' \quad dom(M'') = \{p, \ldots, p + n - 1\}}{malloc(n) : (E, M, P) \twoheadrightarrow (E, M', P[p \mapsto n])} \qquad \frac{E(v_2) \notin dom(M)}{v_1 := *v_2 : (E, M, P) \twoheadrightarrow abort}$$

$$\frac{M = M' \oplus M'' \quad dom(M'') = \{p, \ldots, p + n - 1\} \quad P = P' \cup \{(p, n)\}}{free(n) : (E, M, P) \twoheadrightarrow (E, M', P')}$$

$$\frac{c_1 : (E, M, P) \twoheadrightarrow abort}{c_1; c_2 : (E, M, P) \twoheadrightarrow abort}$$

$$\frac{c_1 : (E, M, P) \twoheadrightarrow (E'', M'', P'') \quad c_2 : (E'', M'', P'') \twoheadrightarrow state}{c_1; c_2 : (E, M, P) \twoheadrightarrow state}$$

$$\frac{\lessdot o \gtrdot E = true \quad c_t : (E, M, P) \twoheadrightarrow state}{if\ o\ then\ c_t\ else\ c_f : (E, M, P) \twoheadrightarrow state} \qquad \frac{\lessdot b \gtrdot E = false \quad c_f : (E, M, P) \twoheadrightarrow state}{if\ o\ then\ c_t\ else\ c_f : (E, M, P) \twoheadrightarrow state}$$

$$\frac{\lessdot c \gtrdot E = false}{while\ o\ do\ c_t : (E, M, P) \twoheadrightarrow (E, M, P)} \qquad \frac{\lessdot o \gtrdot E = true \quad c_t : (E, M, P) \twoheadrightarrow abort}{while\ o\ do\ c_t : (E, M, P) \twoheadrightarrow abort}$$

$$\frac{\lessdot o \gtrdot E = true \quad c_t : (E, M, P) \twoheadrightarrow (E'', M'', P'') \quad while\ o\ do\ c_t : (E'', M'', P'') \twoheadrightarrow state}{while\ o\ do\ c_t : (E, M, P) \twoheadrightarrow state}$$

$$\frac{(\exists\ \xi : n \rightarrow n).\ c_{\xi(1)}; c_{\xi(2)}; \ldots; c_{\xi(n)} : (E, M, P) \twoheadrightarrow state}{par\{\{c_1\}, \ldots, \{c_n\}\} : (E, M, P) \twoheadrightarrow state}$$

$$\frac{par\{\{if\ o_1\ then\ c_1\ else\ skip\}, \ldots, \{if\ o_n\ then\ c_n\ else\ skip\}\} : (E, M, P) \twoheadrightarrow state}{par\text{-}if\{(o_1, c_1), \ldots, (o_n, c_n)\} : (E, M, P) \twoheadrightarrow state}$$

$$\frac{\exists k.\ par\{\overbrace{\{c\}, \ldots, \{c\}}^{k-times}\} : (E, M, P) \twoheadrightarrow state}{par\text{-}for\{c\} : (E, M, P) \twoheadrightarrow state}$$

$$\frac{p' = min\{p \mid \{p, \ldots, p + n - 1\} \cap dom(M) = \emptyset\} \quad \begin{array}{l} M' = M \otimes M'' \\ dom(M'') = \{p', \ldots, p' + n - 1\} \end{array}}{malloc(n) : (E, M, P) \twoheadrightarrow (E, M', P[p' \mapsto n])}$$

Fig. 3. Semantics of Statements of *Core-Par-C*

Theorem 1. *The property of memory safety of terminating and non-terminating programs of the language Core-Par-C is undecidable.*

Proof. We suppose that memory-safety is decidable towards a contradiction. Suppose that $\psi(n)$ is a decidable attribute of natural numbers ($n \in \mathbb{N}$). The attribute $\psi(n)$ can be encrypted by a terminating memory-safe program ($P \equiv x := n; S$) of *Core-Par-C* which is Turing complete. A variable r cab be use in P such that $\psi(n)$ is true (does not hold) if and only if the execution of P ends in a state whose environment assigns 1 (0) to r. Now we consider the program

$$P' \equiv x := 0; \textit{par-for}\{x := x + 1\}; \textit{if } (r = 1) \textit{ then safe else unsafe}.$$

Clearly P' is terminating. Moreover, P' is memory safe if and only if r contains 1 whenever the program terminates. This amounts to the correctness of the attribute ψ for all natural numbers. This is a contradiction because by [15] there exists an attribute ψ such that the attribute $\forall n(\psi(n))$ is proper co-recursively-enumerable.

The details of the proof of Theorem 1 makes it clear that there are several sources of memory un-safety in the *Core-Par-C* programs. These sources are the semantics of the command *malloc* and that of the parallel commands:*par*, *par-for*, and *par-for*. A careful study of the problem reveals that there are restricted versions of these commands semantics that improves the memory-safety characteristics of the Core-Par-C programs. Figure 4 introduces these restricted semantics rules.

Definition 4 introduces the formal definition of *conservatively memory-safety* of *Core-Par-C* programs.

Definition 4. *A program in Core-Par-C is conservatively terminating if it has an execution path in the memory-safe restricted semantics (Figures 2 and 4) that*

$$\frac{c_1; c_2; \ldots; c_n : (E, M, P) \twoheadrightarrow \textit{state}}{\textit{par}\{\{c_1\}, \ldots, \{c_n\}\} : (E, M, P) \twoheadrightarrow \textit{state}}$$

$$\frac{\textit{if } o_1 \textit{ then } c_1 \textit{ else skip}; \ldots; \textit{if } o_n \textit{ then } c_n \textit{ else skip} : (E, M, P) \twoheadrightarrow \textit{state}}{\textit{par-if}\{(o_1, c_1), \ldots, (o_n, c_n)\} : (E, M, P) \twoheadrightarrow \textit{state}}$$

$$\frac{\textit{par}\{\overbrace{\{c\}, \ldots, \{c\}}^{\mu - times}\} : (E, M, P) \twoheadrightarrow \textit{state}}{\textit{par-for}\{c\} : (E, M, P) \twoheadrightarrow \textit{state}}$$

$$\frac{p' = \max(\textit{dom}(M)) \qquad \begin{array}{l} M' = M \otimes M'' \\ \textit{dom}(M'') = \{p', \ldots, p' + n - 1\} \end{array}}{\textit{malloc}(n) : (E, M, P) \twoheadrightarrow (E, M', P[p' \mapsto n])}$$

Fig. 4. Memory-Safe Restricted Semantics of Some Statements of *Core-Par-C*

Operation	Binary code
Allocation	00
Disposal	01
Mutation	10
Looking up	11

Fig. 5. Codes of The Main Four Operations of Separation Logic

does not lead to an abort. A program in Core-Par-C is conservatively memory-safe if for all its possible execution pathes, in the memory-safe restricted semantics, it is conservatively terminating.

Theorem 2 uses Definitions 4 to introduce formally the decidability of *conservatively memory-safety* of *conservatively terminating Core-Par-C* programs.

Theorem 2. *The property of conservatively memory safety of conservatively terminating programs of the language Core-Par-C is decidable.*

The proof of Theorem 2 is by contradiction and is similar to that of Theorem 1.

3 Memory Checkers Using Digital Sequential Circuits

Many areas such as artificial intelligence and systems programming use shared mutable data structures. Such structures are typically mutable in the sense that there are many points for referencing and updating the fields of the data structure. For four decades techniques for reasoning about this approach have been researched. Most of the existing software methods that carrying this reasoning process are either extremely complex or not applicable even to code of moderate length. Very little research were done to achieve such reasoning process using hardware. However such hardware seems like the convenient solution for the complexity and salability issues.

More preciously, this section can be realized as a first attempt to achieve the separation logic as a reasoning tool for shared mutable data structures using digital sequential circuites. The separation logic is built on main four type of commands: allocation, disposal, mutation, and looking up. Therefore our technique starts by code these operations as in Figure 5. In separation logic the memory is reasoned about using four states of memory: empty heap, singleton heap, separation heap, error. The singleton memory has a single allocated cell and the error memory is a memory being treated illegally (may be under attack). The separation heap denotes the states where many septated cells are allocated in the memory.

Figure 7 presents a state diagram explaining effects of four main commands on the different memory states. Different memory states are represented by the nodes of the diagram. The the first two digits of the arc labels (denoted later by x and y) represent the command that transfers the memory from the source

State	Symbol	Binary code
Empty heap	S_0	00
Singleton heap	S_1	01
Separation heap	S_2	10
Error	S_3	11

Fig. 6. Main Four States of Memory and Their Codes

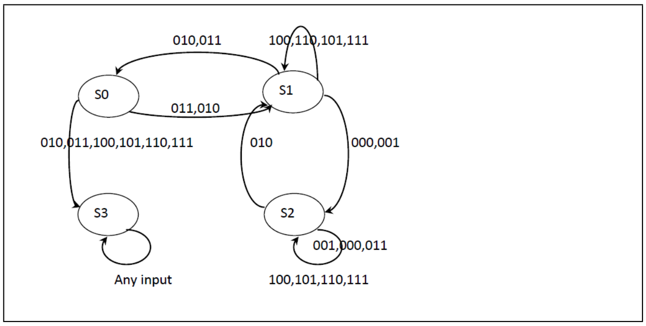


Fig. 7. State Diagram for Effects of Commands on Memory States

state to the target one. The third digit denoted by (MA) is a memory abstraction where 0 means that number of memory cells ≤ 2 and 1 means that number of memory cells > 2.

Figure 8 introduces the truth table of the state diagram of Figure 7.

From the truth table of Figure 8, we can conclude that new states of the memory can be represented as a function of the old states and the inputs which are the command to be executed and the memory abstraction. More precisely, the first column representing the new state can be described by the following equations:

$$A(t+1) = D_A(A, B, x, y, z) =$$

$$\sum(2,3,4,5,6,7,8,9,16,17,19,20,21,22,23,24,25,26,27,28,29,30,31) =$$

$$\prod(0,1,10,11,12,13,14,15,18).$$

This can be represented using the Karnaugh map of Figure 9 which produces the following equation:

$$A(t+1) = D_A(A, B, x, y, z) = AB + Az + Ay' + y'Bx'.$$

The second column representing the new state can be described by the following equations:

Present state		Input			Next state	
		command	code	MA		
A	B	x	y	z	A	B
0	0	0	0	0	0	1
0	0	0	0	1	0	1
0	0	0	1	0	1	1
0	0	0	1	1	1	1
0	0	1	0	0	1	1
0	0	1	0	1	1	1
0	0	1	1	0	1	1
0	0	1	1	1	1	1
0	1	0	1	0	0	0
0	1	0	1	1	0	0
0	1	1	0	0	0	1
0	1	1	0	1	0	1
0	1	1	1	0	0	1
0	1	1	1	1	0	1
0	1	0	0	0	1	0
0	1	0	0	1	1	0
1	0	0	1	0	0	1
1	0	0	0	0	1	0
1	0	0	0	1	1	0
1	0	0	1	1	1	0
1	0	1	0	0	1	0
1	0	1	0	1	1	0
1	0	1	1	0	1	0
1	0	1	1	1	1	0
1	1	x	x	x	1	1

Fig. 8. The Truth Table of The Sequential Circuit Representing the State Diagram

$$B(t+1) = D_A(A,B,x,y,z) =$$

$$\sum(0,1,2,3,4,5,6,7,12,13,14,15,18,24,25,26,27,28,29,30,31) =$$

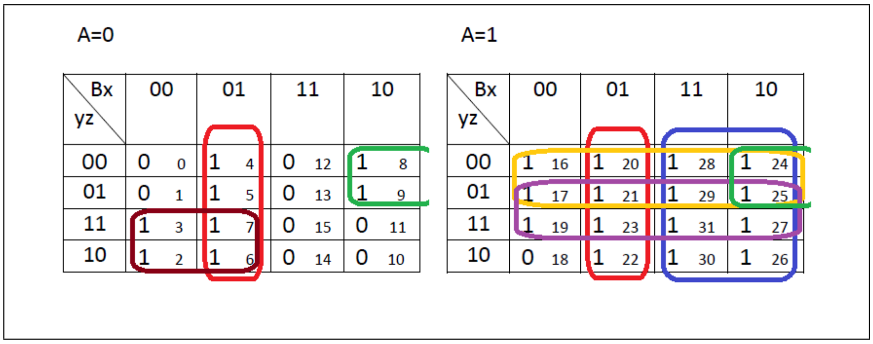
$$\prod(8,9,10,11,16,17,19,20,21,22,23).$$

This can be represented using the Karnaugh map of Figure 10 which produces the following equation:

$$B(t+1) = D_A(A,B,x,y,z) = A'B' + A'x + AB + yz'B'x'.$$

Now a corresponding digital sequential circuit that respects the truth table of Figure 8 and the equations $A(t+1)$ and $B(t+1)$ can be built as in Figure 11. This circuit was build in *Logisim* using two JK flip-flops. The circuit was tested for all cases and its correctness was approved.

All in all, what we have presented in this section is a design for a hardware that is capable of acting as a memory checker against some memory errors. The



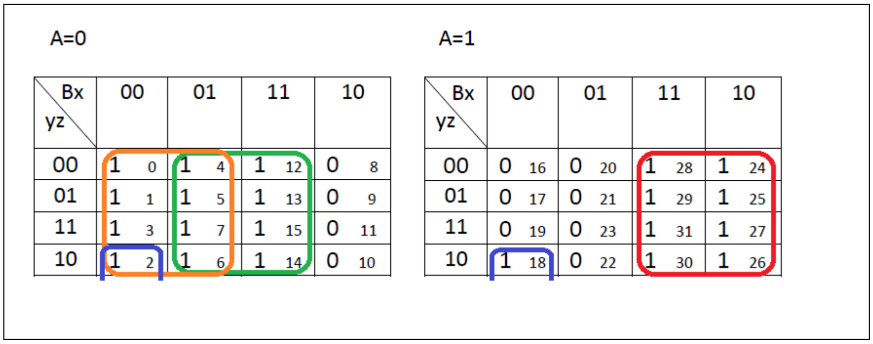
A=0

Bx / yz	00	01	11	10
00	0 0	1 4	0 12	1 8
01	0 1	1 5	0 13	1 9
11	1 3	1 7	0 15	0 11
10	1 2	1 6	0 14	0 10

A=1

Bx / yz	00	01	11	10
00	1 16	1 20	1 28	1 24
01	1 17	1 21	1 29	1 25
11	1 19	1 23	1 31	1 27
10	0 18	1 22	1 30	1 26

Fig. 9. The First Memory Checker State Diagram



A=0

Bx / yz	00	01	11	10
00	1 0	1 4	1 12	0 8
01	1 1	1 5	1 13	0 9
11	1 3	1 7	1 15	0 11
10	1 2	1 6	1 14	0 10

A=1

Bx / yz	00	01	11	10
00	0 16	0 20	1 28	1 24
01	0 17	0 21	1 29	1 25
11	0 19	0 23	1 31	1 27
10	1 18	0 22	1 30	1 26

Fig. 10. The Second Memory Checker State diagram

hardware has the form of a digital sequential circuit. Basic operations behind the design of our proposed hardware are that used in presenting the separation logic. Also main memory states modeled in our design are that considered in separation logic. Therefore this hardware is the main step towards designing a digital sequential circuit to carry the verifications of separation logic.

4 Related and Future Work

This sections reviews work most related to our work. The section also discusses directions for future work.

Much research discuss the fact that a main source of unreliability in programs is violations related to memory access [10,19,31]. Research enumerates problems due to such violations. To avoid such problems many programming languages (such as C++) dynamically detect memory errors via software checks augmented to the programming language. Common disadvantages of the software checks include the reliability on inconvenient metadata, focussing on specific errors, execution overheads, and the reliability on manually changing the code [1,6,30].

Robustness is not paid enough attention compared to quick allocation with minimum fragmentation in most runtime systems. Heap corruption and double

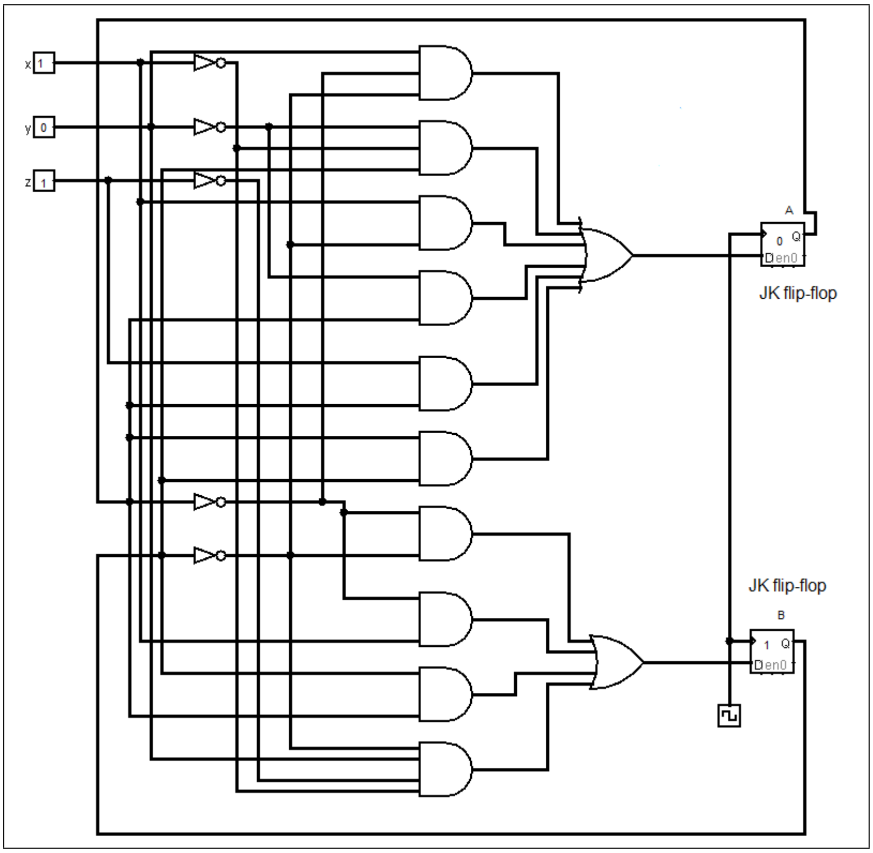


Fig. 11. The Memory Checker circuit

frees due to buffer overflows are caused by most coding applications of malloc; this is true even for the famous GNU C's library. Certain methods are used by some classic garbage collectors and memory controllers [3,27] to support the software robustness. Much more time is needed for reasonably achieving garbage collection than that required by malloc/free [13,36]. Techniques such as DieHard [5] avoids overwrites and invalid and double frees via separating heap from metadata. However DieHard [5] probabilistically (rather than absolutely) avoids dangling references. One more advantage of DieHard [5] over similar techniques is that it discovers unauthorized reads and protects heap content from buffer problems [1,20].

In [29] a static analysis and transformation, *MemSafe*, for guaranteeing protecting the memory safety of C is presented. *MemSafe* can be realized as a technique for casting temporal errors in the form of spatial errors. Merging characteristics of pointer- and object-based procedures, *MemSafe* provides a convenient representation for metadata. *MemSafe* provides a simple and optimal data-flow representation removing unnecessary software checks. However

MemSafe does not treat multi-core programs; it is built for single-core programs [7,32].

Several unsound methods [9,25,27] for preventing memory crashes in programs have been proposed. The idea of automatic pool allocation is to separate regions of memory into pools sharing the same type. This guarantees that dangling references are always replaced only by items of the same type [9]. Unpredictable safely-typed programs result from this method. It is possible to avoid artificial values and illegitimate modifications for manipulation of unprotected regions [27]. Most of these methods significantly increase the performance overhead which may result in faulty program executions. Some methods [25] repair distinguishable errors via using logging and checkpointing together with a file system. This is done via rolling back the software and employees an allocator to avoid defers frees, double frees, pads object requests, and zero-fills buffers [25]. Therefore rolling back techniques [25] are not convenient for softwares with unroll back-based modifications. The fact that rolling back techniques [25] cannot disclose inherent problems resulting in crashes and faulty program executions makes them unsound [11,17].

For C-like programs, a class of memory and type safety approaches [2,18,22,33,35] ends program execution when discovering an error. Such approaches, like *Cyclone* and *CCured*, are called *fail-stop*. In *Cyclone*, programmers have an explicit and secure control over memory via a revised accurate type system attached to C [14]. *Cyclone* is classified as region-based memory management technique [12]. In *CCured*, the code is protected with dynamic test to guarantee memory safety. This technique also employees static analysis to get rid of tests at programs points that are guaranteed to be error-free [22]. To prevent dangling references and double frees, *CCured* uses a garbage collector. Concerning the underlaying program form, DieHard works with binaries and Cyclone, and CCured work with augmented versions of the source code. These augmentations are typically manually made.

The work in [28] focuses on C as the most popular programming tool to implement imperative systems. The low-level memory access provided by C via high-level abstractions and types makes it a perfect object of treating memory problems. The work in [28] relied on the facts that C enables casting, pointer arithmetic, and memory allocation and deallocation. This is very important to consider as such activities are not easy to use which leads to program bugs and security vulnerabilities such as dangling references and stacks overflows. Typically, memory safety of a program means that memory access errors never occur at runtime. In [28] memory safety is treated as the restrictive strict definition applicable for dynamic verifications. In [28], it is shown that generally checking memory safety is undecidable for C programs, as well for terminating closed programs. However, using a restricted concept of memory safety, [28] shows that dynamic verifications of C programs is decidable.

Many techniques were designed to detect both temporal and spatial memory errors. Protecting safety of heap-allocated objects and working on binaries, [35] is one of such techniques. Although focussing on store operations, the approach proposed in [35] improves the detection cost via the use of static analysis. Although being incompatible as a result of using fat-pointers, the technique presented in [2], *Safe C*, protects complete memory safety. Other techniques, such as [24], approaches the memory safety via establishing, in separate processes, separate performing checks and meta-data. The main disadvantage of such techniques is the need for additional CPU power.

In [21], type systems were used to reduce meta-data recording and eliminate the need to check pointer safety. This technique of [21] relies heavily on the use of fat-pointers which resulted in the need for code modifications and in compatibility problems. On the other hand the similar technique of [33] is more efficient and sound although suffering from issues concerting dealing with down casts. The issue with the yet similar technique of [23] is a serious runtime overhead. However the technique of [23,29] is conveniently compatible with ANSI C. Utilizing a characteristic hardware, [8] presents a robust procedure to detect memory bugs.

Developing similar techniques to dynamically study the memory safety of other programming techniques like context-oriented programs and quantum programs is an interesting direction for future work. Another direction for future work is to develop a denotational semantics for dynamically checking the memory safety of programs. An important direction for future work is to develop the logic design of Section 3 to carry the memory verifications of separation logic.

5 Summary

This paper presented an accurate semantics which is employed to mathematically prove the undecidability of memory-safety of a multi-threaded model of programming (*Core-Par-C*).

The paper also proposed a design for a hardware that is capable of acting as a memory checker against some memory errors. The hardware is of the form of digital sequential circuit. Basic operations used in presenting the separation logic are that behind the design of our proposed hardware. Also main memory states considered in separation logic are that modeled in our design. Hence this hardware is a main step in the way to design a digital sequential circuit to achieve the separation-logic verifications.

References

1. Abdulla, P.A., Dwarkadas, S., Rezine, A., Shriraman, A., Zhu, Y.: Verifying safety and liveness for the flextm hybrid transactional memory. In: Macii, E. (ed.) Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18–22, 2013, pp. 785–790. EDA Consortium San Jose, CA, USA / ACM DL (2013)
2. Austin, T.M., Breach, S.E., Sohi, G.S.: Efficient detection of all pointer and array access errors. In: Sarkar, V., Ryder, B.G., Soffa, M.L. (eds.) PLDI, pp. 290–301. ACM (1994)
3. Baker, J., Cunei, A., Kalibera, T., Pizlo, F., Vitek, J.: Accurate garbage collection in uncooperative environments revisited. Concurrency and Computation: Practice and Experience **21**(12), 1572–1606 (2009)

4. Bensalem, S., Peled, D. (eds): Runtime Verification. 9th International Workshop, RV 2009, Grenoble, France, June 26–28, 2009. Selected Papers, volume 5779 of Lecture Notes in Computer Science. Springer (2009)
5. Berger, E.D., Zorn, B.G.: Diehard: probabilistic memory safety for unsafe languages. In: Schwartzbach, M.I., Ball, T. (eds) PLDI, pp. 158–168. ACM (2006)
6. Chatterjee, K., Prabhu, V.S.: Synthesis of memory-efficient, clock-memory free, and non-zeno safety controllers for timed systems. Inf. Comput. **228**, 83–119 (2013)
7. Damm, W., Dierks, H., Oehlerking, J., Pnueli, A.: Towards Component Based Design of Hybrid Systems: Safety and Stability. In: Manna, Z., Peled, D.A. (eds.) Time for Verification. LNCS, vol. 6200, pp. 96–143. Springer, Heidelberg (2010)
8. Dhurjati, D., Adve, V.S.: Efficiently detecting all dangling pointer uses in production servers. In: DSN, pp. 269–280. IEEE Computer Society (2006)
9. Dhurjati, D., Kowshik, S., Adve, V.S., Lattner, C.: Memory safety without runtime checks or garbage collection. In: Mueller, F., Kremer, U. (eds.) LCTES, pp. 69–80. ACM (2003)
10. Dillig, T., Dillig, I., Chaudhuri, S.: Optimal guard synthesis for memory safety. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 491–507. Springer, Heidelberg (2014)
11. Godefroid, P., Kinder, J.: Proving memory safety of floating-point computations by combining static and dynamic program analysis. In: Tonella, P., Orso, A. (eds.), Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12–16, 2010, pp. 1–12. ACM (2010)
12. Grossman, D., Morrisett, J.G., Jim, T., Hicks, M.W., Wang, Y., Cheney, J.: Region-based memory management in cyclone. In: Knoop, J., Hendren, L.J. (eds.) PLDI, pp. 282–293. ACM (2002)
13. Hertz, M., Berger, E.D.: Quantifying the performance of garbage collection vs. explicit memory management. In: Johnson, R.E., Gabriel, R.P. (eds.) OOPSLA, pp. 313–326. ACM (2005)
14. Jim, T., Morrisett, J.G., Grossman, D., Hicks, M.W., Cheney, J., Wang, Y.: Cyclone: A safe dialect of c. In: Ellis, C.S. (ed.) USENIX Annual Technical Conference, General Track, pages 275–288. USENIX (2002)
15. Rogers Jr, H.: Theory of Recursive Functions and Effective Computability. MIT press, Cambridge, MA (1987)
16. Lamport, L.: The hoare logic' of concurrent programs. Acta Inf. **14**, 21–37 (1980)
17. Lee, H.-C., Seong, P.-H.: A computational model for evaluating the effects of attention, memory, and mental models on situation assessment of nuclear power plant operators. Rel. Eng. & Sys. Safety **94**(11), 1796–1805 (2009)
18. Li, H., Gao, H., Shi, P., Zhao, X.: Fault-tolerant control of markovian jump stochastic systems via the augmented sliding mode observer approach. Automatica **50**(7), 1825–1834 (2014)
19. Marriott, C., Cavalcanti, A.: SCJ: memory-safety checking without annotations. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 465–480. Springer, Heidelberg (2014)
20. Nagarakatte, S., Martin, M.M.K., Zdancewic, S.: Watchdog: Hardware for safe and secure manual memory management and full memory safety. In: 39th International Symposium on Computer Architecture (ISCA 2012), June 9–13, 2012, Portland, OR, USA, pp. 189–200. IEEE (2012)
21. Necula, G.C., Condit, J., Harren, M., McPeak, S., Weimer, W.: Ccured: type-safe retrofitting of legacy software. ACM Trans. Program. Lang. Syst. **27**(3), 477–526 (2005)

22. Necula, G.C., McPeak, S., Weimer, W.: Ccured: type-safe retrofitting of legacy code. In: Launchbury, J., Mitchell, J.C. (eds.) POPL, pp. 128–139. ACM (2002)
23. Oiwa, Y.: Implementation of the memory-safe full ansi-c compiler. In: Hind, M., Diwan, A. (eds) PLDI, pp. 259–269. ACM (2009)
24. Patil, H., Fischer, C.N.: Low-cost, concurrent checking of pointer and array accesses in c programs. Softw., Pract. Exper. **27**(1), 87–110 (1997)
25. Qin, F., Tucek, J., Zhou, Y., Sundaresan, J.: Rx: Treating bugs as allergies - a safe method to survive software failures. ACM Trans. Comput. Syst. **25**(3) (2007)
26. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22–25 July 2002, Copenhagen, Denmark, Proceedings, pp. 55–74. IEEE Computer Society (2002)
27. Rinard, M.C., Cadar, C., Dumitran, D., Roy, D.M., Leu,T., Beebee, W.S.: Enhancing server availability and security through failure-oblivious computing. In: Brewer, E.A., Chen, P. (eds.) OSDI, pp. 303–316. USENIX Association (2004)
28. Rosu, G., Schulte, W., Serbanuta, T.-F.: Runtime verification of c memory safety. In: Bensalem and Peled [4], pp. 132–151
29. Simpson, M.S., Barua, R.: Memsafe: ensuring the spatial and temporal memory safety of c at runtime. Softw., Pract. Exper. **43**(1), 93–128 (2013)
30. Singh, A., Narayanasamy, S., Marino, D., Millstein, T.D., Musuvathi, M.: A safety-first approach to memory models. IEEE Micro **33**(3), 96–104 (2013)
31. Ströder, T., Giesl, J., Brockschmidt, M., Frohn, F., Fuhs, C., Hensel, J., Schneider-Kamp, P.: Proving termination and memory safety for programs with pointer arithmetic. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS, vol. 8562, pp. 208–223. Springer, Heidelberg (2014)
32. Vazou, N., Papakyriakou, M.A., Papaspyrou, N.: Memory safety and race freedom in concurrent programming languages with linear capabilities. In: Ganzha, M., Maciaszek, L.A., Paprzycki, M. (eds.) Federated Conference on Computer Science and Information Systems - FedCSIS 2011, Szczecin, Poland, 18–21 September 2011, Proceedings, pp. 833–840 (2011)
33. Xu, W., DuVarney, D.C., Sekar, R.: An efficient and backwards-compatible transformation to ensure memory safety of c programs. In: Taylor, R.N., Dwyer, M.B. (eds.) SIGSOFT FSE, pp. 117–126. ACM (2004)
34. Yang, J., Cui, H., Jingyue, W., Tang, Y., Gang, H.: Making parallel programs reliable with stable multithreading. Commun. ACM **57**(3), 58–69 (2014)
35. Yong, S.H., Horwitz, S.: Protecting c programs from attacks via invalid pointer dereferences. In: ESEC / SIGSOFT FSE, pp. 307–316. ACM (2003)
36. Zorn, B.G.: The measured cost of conservative garbage collection. Softw., Pract. Exper. **23**(7), 733–756 (1993)