Certified Self-Modifying Code

Hongxu Cai
Department of Computer Science and Technology, Tsinghua University
Beijing, 100084, China
hxcai00@mails.tsinghua.edu.cn

Zhong Shao
Department of Computer Science
Yale University
New Haven, CT 06520, USA
shao@cs.yale.edu

Alexander Vaynberg
Department of Computer Science
Yale University
New Haven, CT 06520, USA
alv@cs.yale.edu

Abstract

Self-modifying code (SMC), in this paper, broadly refers to any program that loads, generates, or mutates code at runtime. It is widely used in many of the world's critical software systems to support runtime code generation and optimization, dynamic loading and linking, OS boot loader, just-in-time compilation, binary translation, or dynamic code encryption and obfuscation. Unfortunately, SMC is also extremely difficult to reason about: existing formal verification techniques—including Hoare logic and type system—consistently assume that program code stored in memory is fixed and immutable; this severely limits their applicability and power.

This paper presents a simple but novel Hoare-logic-like framework that supports modular verification of general von-Neumann machine code with runtime code manipulation. By dropping the assumption that code memory is fixed and immutable, we are forced to apply local reasoning and separation logic at the very beginning, and treat program code uniformly as regular data structure. We address the interaction between separation and code memory and show how to establish the frame rules for local reasoning even in the presence of SMC. Our framework is realistic, but designed to be highly generic, so that it can support assembly code under all modern CPUs (including both x86 and MIPS). Our system is expressive and fully mechanized. We prove its soundness in the Coq proof assistant and demonstrate its power by certifying a series of realistic examples and applications—all of which can directly run on the SPIM simulator or any stock x86 hardware.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—correctness proofs, formal methods; D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Verification

Keywords self-modifying code, runtime code manipulation, assembly code verification, modular verification, Hoare logic

1. Introduction

Self-modifying code (SMC), in this paper, broadly refers to any program that purposely loads, generates, or mutates code at runtime. It is widely used in many of the world's critical software systems. For example, runtime code generation and compilation can improve the performance of operating systems [20] and other application programs [19, 13, 29]. Dynamic code optimization can improve the performance [4, 11] or minimize the code size [7]. Dynamic code encryption [27] or obfuscation [15] can support code protection and tamper-resistant software [3]; they also make it hard for crackers to debug or decompile the protected binaries. Evolutionary computing systems can use dynamic techniques to support genetic programming [25]. SMC also arises in applications such as just-in-time compiler, dynamic loading and linking, OS bootloaders, binary translation, and virtual machine monitor.

	Examples	System	where
Common Basic Constructs	opcode modification	GCAP2	Sec 5
	control flow modification	GCAP2	TR[5]
	unbounded code rewriting	GCAP2	Sec 6.2
	runtime code checking	GCAP1	TR[5]
	runtime code generation	GCAP1	TR[5]
	multilevel RCG	GCAP1	Sec 4
	self-mutating code block	GCAP2	TR[5]
	mutual modification	GCAP2	TR[5]
Typical Applications	self-growing code	GCAP2	Sec 6.3
	polymorphic code	GCAP2	TR[5]
	code optimization	GCAP2/1	TR[5]
	code compression	GCAP1	TR[5]
	code obfuscation	GCAP2	TR[5]
	code encryption	GCAP1	Sec 6.4
	OS boot loaders	GCAP1	Sec 6.1
	shellcode	GCAP1	TR[5]

Table 1. A summary of GCAP-supported applications

Unfortunately, SMC is also extremely difficult to reason about: existing formal verification techniques—including Hoare logic [8, 12] and type system [26, 22]—consistently assume that program code stored in memory is immutable; this significantly limits their power and applicability.

In this paper we present a simple but powerful Hoare-logic-style framework—namely GCAP (i.e., CAP [31] on General von Neumann machines)—that supports modular verification of general machine code with runtime code manipulation. By dropping the assumption that code memory is fixed and immutable, we are forced to apply local reasoning and separation logic [14, 28] at the very beginning, and treat program code uniformly as regular data structure. Our framework is realistic, but designed to be highly generic, so that it can support assembly code under all modern CPUs (including both x86 and MIPS). Our paper makes the following new contributions:

- Our GCAP system is the first formal framework that can successfully certify any form of runtime code manipulation—

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'07 June 11–13, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-633-2/07/0006. . . $5.00

$$
\begin{array}{rrl}
(\textit{Machine}) & \mathcal{M} & ::= (\textit{Extension}, \textit{Instr}, \mathsf{Ec} : \textit{Instr} \to \textit{ByteList}, \\
 & & \quad \mathsf{Next} : \textit{Address} \to \textit{Instr} \to \textit{State} \rightharpoonup \textit{State}, \\
 & & \quad \mathsf{Npc} : \textit{Address} \to \textit{Instr} \to \textit{State} \to \textit{Address}) \\
(\textit{State}) & \mathbb{S} & ::= (\mathbb{M}, \mathbb{E}) \\
(\textit{Mem}) & \mathbb{M} & ::= \{\mathtt{f} \rightsquigarrow \mathtt{b}\}^* \\
(\textit{Extension}) & \mathbb{E} & ::= \ldots \\
(\textit{Address}) & \mathtt{f}, \mathtt{pc} & ::= \ldots \quad (\textit{nat nums}) \\
(\textit{Byte}) & \mathtt{b} & ::= \ldots \quad (0..255) \\
(\textit{ByteList}) & \mathtt{bs} & ::= \mathtt{b}, \mathtt{bs} \mid \mathtt{b} \\
(\textit{Instr}) & \iota & ::= \ldots \\
(\textit{World}) & \mathbb{W} & ::= (\mathbb{S}, \mathtt{pc})
\end{array}
$$

Figure 1. Definition of target machine GTM

including all the common basic constructs and important applications given in Table 1 (due to the space limit, we have to leave many examples in the companion TR [5]).We are the first to successfully certify a realistic OS boot loader that can directly boot on stock x86 hardware. All of our MIPS examples can be directly executed in the SPIM 7.3 simulator[17].

- GCAP is the first successful extension of Hoare-style program logic that treats machine instructions as regular mutable data structures. A general GCAP assertion can not only specify the usual precondition for data memory but also can ensure that code segments are correctly loaded into memory before execution. We develop the idea of *parametric code blocks* to specify and reason about all possible outputs of each self-modifying program. These results are general and can be easily applied to other Hoare-style verification systems.
- GCAP supports both modular verification [9] and frame rules for local reasoning [28]. Program modules can be verified separately and with minimized import interfaces. GCAP pinpoints the precise boundary between non-self-modifying code groups and those that do manipulate code at runtime. Non-self-modifying code groups can be certified without any knowledge about each other's implementation, yet they can still be safely linked together with other self-modifying code groups.
- GCAP is highly generic in the sense that it is the first attempt to support different machine architectures and instruction sets in the same framework without modifying any of its inference rules. This is done by making use of several auxiliary functions that abstract away the machine-specific semantics and by constructing generic (platform-independent) inference rules for certifying well-formed code sequences.

In the rest of this paper, we first present our von-Neumann machine GTM in Section 2. We stage the presentation of GCAP: Section 3 presents a Hoare-style program logic for GTM; Section 4 presents a simple extended GCAP1 system for certifying runtime code loading and generation; Section 5 presents GCAP2 which extends GCAP1 with general support of SMC. In Section 6 and in the companion TR [5], we present a large set of certified SMC applications to demonstrate the power and practicality of our framework. Our system is fully mechanized—the Coq implementation (including the full soundness proof) is available on the web [5].

2. General Target Machine GTM

Our general machine model, namely GTM, is an abstract framework for von Neumann machines. GTM is general because it can be used to model modern computing architecture such as x86, MIPS, or PowerPC. Fig 1 shows the essential elements of GTM. An instance $\mathcal{M}$ of a GTM machine is modeled as a 5-tuple that determines the machine's operational semantics.

If $\mathtt{Decode}(\mathbb{S}, \mathtt{pc}, \iota)$ is true, then

$$(\mathbb{S}, \mathtt{pc}) \longmapsto \left(\mathsf{Next}_{\mathtt{pc},\iota}(\mathbb{S}),\ \mathsf{Npc}_{\mathtt{pc},\iota}(\mathbb{S})\right)$$

Figure 2. GTM program execution

A machine state $\mathbb{S}$ should consist of at least a memory component $\mathbb{M}$, which is a partial map from the memory address to its stored *Byte* value. *Byte* specifies the machine byte which is the minimum unit of memory addressing. Note that because the memory component is a partial map, its domain can be any subset of natural numbers. $\mathbb{E}$ represents other additional components of a state, which may include register files and disks, etc. No explicit code heap is involved: all the code is encoded and stored in the memory and can be accessed just as regular data. *Instr* specifies the instruction set, with an encoding function Ec describing how instructions can be stored in memory as byte sequences. We also introduce an auxiliary Decode predicate which is defined as follows:

$$\mathtt{Decode}((\mathbb{M},\mathbb{E}), \mathtt{f}, \iota) \triangleq \mathsf{Ec}(\iota) = (\mathbb{M}[\mathtt{f}], \ldots, \mathbb{M}[\mathtt{f}+|\mathsf{Ec}(\iota)|-1])$$

In other words, $\mathtt{Decode}(\mathbb{S}, \mathtt{f}, \iota)$ states that under the state $\mathbb{S}$, certain consecutive bytes stored starting from memory address f are precisely the encoding of instruction ι.

Program execution is modeled as a small-step transition relation between two *World*s (i.e., $\mathbb{W} \longmapsto \mathbb{W}'$), where a world $\mathbb{W}$ is just a state plus a program counter pc that indicates the next instruction to be executed. The definition of this transition relation is formalized in Fig 2. Next and Npc are two functions that define the behavior of all available instructions. When instruction ι located at address pc is executed at state $\mathbb{S}$, $\mathsf{Next}_{\mathtt{pc},\iota}(\mathbb{S})$ is the resulting state and $\mathsf{Npc}_{\mathtt{pc},\iota}(\mathbb{S})$ is the resulting program counter. Note that Next could be a partial function (since memory is partial) while Npc is always total.

To make a step, a certain number of bytes starting from pc are fetched out and decoded into an instruction, which is then executed following the Next and Npc functions. There will be no transition if Next is undefined on a given state. As expected, if there is no valid transition from a world, the execution gets stuck.

To make program execution deterministic, the following condition should be satisfied:

$$\forall \mathbb{S}, \mathtt{f}, \iota_1, \iota_2. \mathtt{Decode}(\mathbb{S}, \mathtt{f}, \iota_1) \wedge \mathtt{Decode}(\mathbb{S}, \mathtt{f}, \iota_2) \longrightarrow \iota_1 = \iota_2$$

In other words, Ec should be prefix-free: under no circumstances should the encoding of one instruction be a prefix of the encoding of another one. Instruction encodings on real machines follow regular patterns (e.g., the actual value for each operand is extracted from certain bits). These properties are critical when involving operand-modifying instructions. Appel *et al* [2, 21] gave a more specific decoding relation and an in-depth analysis.

The definitions of the Next and Npc functions should also guarantee the following property: if $((\mathbb{M},\mathbb{E}),\mathtt{pc}) \longmapsto ((\mathbb{M}',\mathbb{E}'),\mathtt{pc}')$ and $\mathbb{M}''$ is a memory whose domain does not overlap with those of $\mathbb{M}$ and $\mathbb{M}'$, then $((\mathbb{M} \cup \mathbb{M}'',\mathbb{E}),\mathtt{pc}) \longmapsto ((\mathbb{M}' \cup \mathbb{M}'',\mathbb{E}'),\mathtt{pc}')$. In other words, adding extra memory does not affect the execution process of the original world.

MIPS specialization. The MIPS machine $\mathcal{M}_{\text{MIPS}}$ is built as an instance of the GTM framework (Fig 3). In $\mathcal{M}_{\text{MIPS}}$, the machine state consists of a $(\mathbb{M},\mathbb{R})$ pair, where $\mathbb{R}$ is a register file, defined as a map from each of the 31 registers to a stored value. \$0 is not included in the register set since it always stores constant zero and is immutable according to MIPS convention. A machine *Word* is the composition of four *Byte*s. To achieve interaction between registers and memory, two operators — $\langle\cdot\rangle_1$ and $\langle\cdot\rangle_4$ — are defined (details omitted here) to do type conversion between *Word* and *Value*.

$$
\begin{array}{rrl}
(\textit{State}) & \mathbb{S} & ::= (\mathbb{M}, \mathbb{R}) \\
(\textit{RegFile}) & \mathbb{R} & \in \textit{Register} \to \textit{Value} \\
(\textit{Register}) & \mathtt{r} & ::= \$1 \mid \ldots \mid \$31 \\
(\textit{Value}) & i, \langle \mathtt{w} \rangle_1 & ::= \ldots \quad (\textit{int nums}) \\
(\textit{Word}) & \mathtt{w}, \langle i \rangle_4 & ::= \mathtt{b}, \mathtt{b}, \mathtt{b}, \mathtt{b} \\
(\textit{Instr}) & \iota & ::= \mathsf{nop} \mid \mathsf{li}\ \mathtt{r}_d, i \mid \mathsf{add}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t \mid \mathsf{addi}\ \mathtt{r}_t, \mathtt{r}_s, i \\
 & & \quad \mid \mathsf{mul}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t \mid \mathsf{lw}\ \mathtt{r}_t, i(\mathtt{r}_s) \mid \mathsf{sw}\ \mathtt{r}_t, i(\mathtt{r}_s) \\
 & & \quad \mid \mathsf{la}\ \mathtt{r}_d, \mathtt{f} \mid \mathsf{j}\ \mathtt{f} \mid \mathsf{jr}\ \mathtt{r}_s \mid \mathsf{beq}\ \mathtt{r}_s, \mathtt{r}_t, i \mid \mathsf{jal}\ \ \mathtt{f}
\end{array}
$$

Figure 3. $\mathcal{M}_{\text{MIPS}}$ data types

$\mathsf{Ec}(\iota) \triangleq \ldots,$

if $\iota =$	then $\mathsf{Next}_{\mathtt{pc},\iota}(\mathbb{M},\mathbb{R}) =$
jal f	$(\mathbb{M}, \mathbb{R}\{\$31 \rightsquigarrow \mathtt{pc}+4\})$
nop	$(\mathbb{M}, \mathbb{R})$
li $\mathtt{r}_d, i$ / la $\mathtt{r}_d, i$	$(\mathbb{M}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow i\})$
add $\mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t$	$(\mathbb{M}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s)+\mathbb{R}(\mathtt{r}_t)\})$
addi $\mathtt{r}_t, \mathtt{r}_s, i$	$(\mathbb{M}, \mathbb{R}\{\mathtt{r}_t \rightsquigarrow \mathbb{R}(\mathtt{r}_s)+i\})$
mul $\mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t$	$(\mathbb{M}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s)\times\mathbb{R}(\mathtt{r}_t)\})$
lw $\mathtt{r}_t, i(\mathtt{r}_s)$	$(\mathbb{M}, \mathbb{R}\{\mathtt{r}_t \rightsquigarrow \langle \mathbb{M}(\mathtt{f}), \ldots, \mathbb{M}(\mathtt{f}+3)\rangle_1\})$ if $\mathtt{f} = \mathbb{R}(\mathtt{r}_s)+i \in \mathsf{dom}(\mathbb{M})$
sw $\mathtt{r}_t, i(\mathtt{r}_s)$	$(\mathbb{M}\{\mathtt{f}, \ldots, \mathtt{f}+3 \rightsquigarrow \langle \mathbb{R}(\mathtt{r}_t)\rangle_4\}, \mathbb{R})$ if $\mathtt{f} = \mathbb{R}(\mathtt{r}_s)+i \in \mathsf{dom}(\mathbb{M})$
Otherwise	$(\mathbb{M}, \mathbb{R})$

and

if $\iota =$	then $\mathsf{Npc}_{\mathtt{pc},\iota}(\mathbb{M},\mathbb{R}) =$
j f	f
jr $\mathtt{r}_s$	$\mathbb{R}(\mathtt{r}_s)$
beq $\mathtt{r}_s, \mathtt{r}_t, i$	$\begin{cases} \mathtt{pc}+i & \text{when } \mathbb{R}(\mathtt{r}_s) = \mathbb{R}(\mathtt{r}_t), \\ \mathtt{pc}+4 & \text{when } \mathbb{R}(\mathtt{r}_s) \neq \mathbb{R}(\mathtt{r}_t) \end{cases}$
jal f	f
Otherwise	$\mathtt{pc}+\lvert\mathsf{Ec}(\iota)\rvert$

Figure 4. $\mathcal{M}_{\text{MIPS}}$ operational semantics

$$
\begin{array}{rrl}
(\textit{Word}) & \mathtt{w} & ::= \mathtt{b}, \mathtt{b} \\
(\textit{State}) & \mathbb{S} & ::= (\mathbb{M}, \mathbb{R}, \mathbb{D}) \\
(\textit{RegFile}) & \mathbb{R} & ::= \{\mathtt{r}^{16} \rightsquigarrow \mathtt{w}\}^* \cup \{\mathtt{r}^s \rightsquigarrow \mathtt{w}\}^* \\
(\textit{Disk}) & \mathbb{D} & ::= \{l \rightsquigarrow \mathtt{b}\}^* \\
(\textit{Word Regs}) & \mathtt{r}^{16} & ::= \mathtt{r}_{AX} \mid \mathtt{r}_{BX} \mid \mathtt{r}_{CX} \mid \mathtt{r}_{DX} \mid \mathtt{r}_{SI} \mid \mathtt{r}_{DI} \mid \mathtt{r}_{BP} \mid \mathtt{r}_{SP} \\
(\textit{Byte Regs}) & \mathtt{r}^8 & ::= \mathtt{r}_{AH} \mid \mathtt{r}_{AL} \mid \mathtt{r}_{BH} \mid \mathtt{r}_{BL} \mid \mathtt{r}_{CH} \mid \mathtt{r}_{CL} \mid \mathtt{r}_{DH} \mid \mathtt{r}_{DL} \\
(\textit{Segment Regs}) & \mathtt{r}^s & ::= \mathtt{r}_{DS} \mid \mathtt{r}_{ES} \mid \mathtt{r}_{SS} \\
(\textit{Instr}) & \iota & ::= \mathsf{movw}\ \mathtt{w}, \mathtt{r}^{16} \mid \mathsf{movw}\ \mathtt{r}^{16}, \mathtt{r}^S \mid \mathsf{movb}\ \mathtt{b}, \mathtt{r}^8 \\
 & & \quad \mid \mathsf{jmp}\ \mathtt{b} \mid \mathsf{jmpl}\ \mathtt{w}, \mathtt{w} \mid \mathsf{int}\ \mathtt{b} \mid \ldots
\end{array}
$$

Figure 5. $\mathcal{M}_{\text{x86}}$ data types

The set of instructions *Instr* is minimal and it contains only the basic MIPS instructions, but extensions can be easily made. $\mathcal{M}_{\text{MIPS}}$ supports direct jump, indirect jump, and jump-and-link (`jal`) instructions. It also provides relative addressing for branch instructions (e.g. beq $\mathtt{r}_s, \mathtt{r}_t, i$), but for clarity we will continue using code labels to represent the branching targets in our examples.

The Ec function follows the official MIPS documentation and is omitted. Interested readers can read our Coq implementation. Fig 4 gives the definitions of Next and Npc. It is easy to see that these functions indeed satisfy the requirements we mentioned earilier.

x86 (16-bit) specialization. In Fig 5, we show our x86 machine, $\mathcal{M}_{\text{x86}}$, as an instance of GTM. The specification of $\mathcal{M}_{\text{x86}}$ is a restriction of the real x86 architecture. However, it is adequate for certification of interesting examples such as OS boot loaders.

$\mathsf{Ec}(\iota) \triangleq \ldots,$

if $\iota =$	then $\mathsf{Next}_{\mathtt{pc},\iota}(\mathbb{M},\mathbb{R},\mathbb{D}) =$
movw w, $\mathtt{r}^{16}$	$(\mathbb{M}, \mathbb{R}\{\mathtt{r}^{16} \rightsquigarrow \mathtt{w}\}, \mathbb{D})$
movw $\mathtt{r}^{16}, \mathtt{r}^S$	$(\mathbb{M}, \mathbb{R}\{\mathtt{r}^S \rightsquigarrow \mathbb{R}(\mathtt{r}^{16})\}, \mathbb{D})$
movb b, $\mathtt{r}^8$	$(\mathbb{M}, \mathbb{R}\{\mathtt{r}^8 \rightsquigarrow \mathtt{b}\}, \mathbb{D})$
jmp b	$(\mathbb{M}, \mathbb{R}, \mathbb{D})$
jmpl $\mathtt{w}_1, \mathtt{w}_2$	$(\mathbb{M}, \mathbb{R}, \mathbb{D})$
int b	BIOS Call b $(\mathbb{M}, \mathbb{R}, \mathbb{D})$
...	...

and

if $\iota =$	then $\mathsf{Npc}_{\mathtt{pc},\iota}(\mathbb{M},\mathbb{R}) =$
jmp b	$\mathtt{pc}+2+\mathtt{b}$
jmpl $\mathtt{w}_1, \mathtt{w}_2$	$\mathtt{w}_1 * 16 + \mathtt{w}_2$
Non-jmp instructions	$\mathtt{pc}+\lvert\mathsf{Ec}(\iota)\rvert$
...	...

Figure 6. $\mathcal{M}_{\text{x86}}$ operational semantics

Call 0x13 (disk operations) (id = 0x13)

Command 0x02 (disk read) ($\mathbb{R}(\mathtt{r}_{AH})$ = 0x02)

Parameters

Count	= $\mathbb{R}(\mathtt{r}_{AL})$	Cylinder	= $\mathbb{R}(\mathtt{r}_{CH})$
Sector	= $\mathbb{R}(\mathtt{r}_{CL})$	Head	= $\mathbb{R}(\mathtt{r}_{DH})$
Disk Id	= $\mathbb{R}(\mathtt{r}_{DL})$	Bytes	= Count $*$ 512
Src	= (Sector – 1) $*$ 512		
Dest	= $\mathbb{R}(\mathtt{r}_{ES}) * 16 + \mathbb{R}(\mathtt{r}_{BX})$		

Conditions

Cylinder = 0	Head = 0
Disk Id = 0x80	Sector < 63

Effect

$\mathbb{M}' = \mathbb{M}\{Dest + i \rightsquigarrow \mathbb{D}(Src + i)\}$ $(0 \le i \le \text{Bytes})$

$\mathbb{R}' = \mathbb{R}\{\mathtt{r}_{AH} \rightsquigarrow 0\}$ $\mathbb{D}' = \mathbb{D}$

Figure 7. Subset of $\mathcal{M}_{\text{x86}}$ BIOS operations

```
    .data   # Data declaration section

100 new:    addi $2, $2, 1        # the new instr

    .text   # Code section

200 main:   beq  $2, $4, modify   # do modification
204 target: move $2, $4           # original instr
208         j    halt             # exit

212 halt:   j    halt

216 modify: lw   $9, new          # load new instr
224         sw   $9, target       # store to target
232         j    target           # return
```

Figure 8. `opcode.s`: Opcode modification

In order to certify a boot loader, we augment the $\mathcal{M}_{\text{x86}}$ state to include a disk. Everything else that is machine specific has no effect on the GTM specification. Operations on the 8-bit registers are done via their corresponding 16-bit registers (see TR [5] for more detail). To use the disk, $\mathcal{M}_{\text{x86}}$ needs to call a firmware such as BIOS, which we treat as a black box with proper formal specifications (Fig 7). We define the BIOS call as a primitive operation in the semantics. In this paper, we only define a BIOS command for disk read, as it is needed for our boot loader. Since we did not want to present a complex definition of the disk, we assume our disk has only one cylinder, one side, and 63 sectors.

A Taste of SMC We give a sample piece of self-modifying code (i.e., `opcode.s`) in Fig 8. The example is written in $\mathcal{M}_{\text{MIPS}}$ syntax.

We use line numbers to indicate the location of each instruction in memory. It seems that this program will copy the value of register \$4 to register \$2 and then call `halt`. But it could jump to the `modify` subroutine first which will overwrite the `target` code with the new instruction addi \$2, \$2, 1. So the actual result of this program can vary: if $\mathbb{R}(\$2) \neq \mathbb{R}(\$4)$, the program copies the value of \$4 to \$2; otherwise, the program simply adds \$2 by 1.

Even such a simple program cannot be handled by any existing verification frameworks since none of them allow code to be mutated at anytime. General SMCs are even more challenging: they are difficult to understand and reason about because the actual program itself is changing during the execution—it is difficult to figure out the program's control and data flow.

3. Hoare-Style Reasoning under GTM

Hoare-style reasoning has always been done over programs with separate code memory. In this section we want to eliminate such restriction. To reason about GTM programs, we formalize the syntax and the operational semantics of GTM inside a mechanized meta logic. For this paper we will use the calculus of inductive constructions (CiC) [30] as our meta logic. Our implementation is done using Coq [30] but all our results also apply to other proof assistants.

We will use the following syntax to denote terms and predicates in the meta logic:

$$(\textit{Term})\ A, B ::= \mathsf{Set} \mid \mathsf{Prop} \mid \mathsf{Type} \mid x \mid \lambda x{:}A.\,B \mid A\ B \mid A \to B \mid \textit{ind. def.} \mid \ldots$$

$$(\textit{Prop})\ p, q ::= \mathsf{True} \mid \mathsf{False} \mid \neg p \mid p \wedge q \mid p \vee q \mid p \to q \mid \forall x{:}A.\,p \mid \exists x{:}A.\,p \mid \ldots$$

The program safety predicate can be defined as follows:

$$\mathsf{Safe}_n(\mathbb{W}) \triangleq \begin{cases} \mathtt{True} & \text{if } n = 0 \\ \exists \mathbb{W}'.\, \mathbb{W} \longmapsto \mathbb{W}' \wedge \mathsf{Safe}_{n-1}(\mathbb{W}') & \text{if } n > 0 \end{cases}$$

$$\mathsf{Safe}(\mathbb{W}) \triangleq \forall n{:}\mathtt{Nat}.\,\mathsf{Safe}_n(\mathbb{W})$$

Safe_n states that the machine is safe to execute n steps from a world, while Safe describes that the world is safe to run forever.

Invariant-based method [16] is a common technique for proving safety properties of programs.

Definition 3.1 An **invariant** is a predicate, namely Inv, defined over machine worlds, such that the following holds:

- $\forall \mathbb{W}.\, \mathsf{Inv}(\mathbb{W}) \longrightarrow \exists \mathbb{W}'.\, (\mathbb{W} \longmapsto \mathbb{W}')$ (Progress)
- $\forall \mathbb{W}.\, \mathsf{Inv}(\mathbb{W}) \wedge (\mathbb{W} \longmapsto \mathbb{W}') \longrightarrow \mathsf{Inv}(\mathbb{W}')$ (Preservation)

The existence of an invariant immediately implies program safety, as shown by the following theorem.

Theorem 3.2 If Inv is an invariant then $\forall \mathbb{W}.\, \mathsf{Inv}(\mathbb{W}) \to \mathsf{Safe}(\mathbb{W})$.

Traditional Hoare-style reasoning over assembly programs (e.g., CAP [31]) is illustrated in Fig 9. Program code is assumed to be stored in a static code heap separated from the main memory. A code heap can be divided into different code blocks (i.e. consecutive instruction sequences) which serve as basic certifying units. A precondition is assigned to every code block, whereas no postcondition shows up since we often use CPS (continuation passing style) to reason about low-level programs. Different blocks can be independently verified then linked together to form a global invariant and complete the verification.

Here we present a Hoare-logic-based system GCAP0 for GTM. Developing a Hoare logic for GTM is not trivial. Firstly, unifying different types of instructions (especially between regular command and control transfer instruction) without loss of usability requires an intrinsic understanding of the relation between instructions and program specifications. Secondly, code is easily guaranteed to be immutable in an abstract machine that separates code heap as an individual component, which GTM is different from. Surprisingly, the same immutability can be enforced in the inference rules using a simple separation conjunction borrowed from separation logic [14, 28].

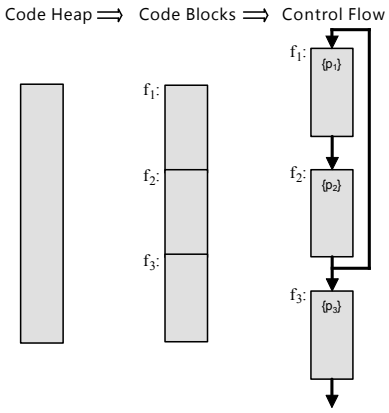

Figure 9. Hoare-style reasoning of assembly code

$$(\textit{CodeBlock})\ \ \mathbb{B} ::= \mathtt{f}{:}\,\mathbb{I}$$
$$(\textit{InstrSeq})\ \ \mathbb{I} ::= \iota; \mathbb{I} \mid \iota$$
$$(\textit{CodeHeap})\ \ \mathbb{C} ::= \{\mathtt{f} \rightsquigarrow \mathbb{I}\}^*$$
$$(\textit{Assertion})\ \ \mathtt{a} \in \textit{State} \to \textit{Prop}$$
$$(\textit{ProgSpec})\ \ \Psi \in \textit{Address} \rightharpoonup \textit{Assertion}$$

Figure 10. Auxiliary constructs and specifications

$$\mathtt{a} \Rightarrow \mathtt{a}' \triangleq \forall \mathbb{S}.(\mathtt{a}\ \mathbb{S} \to \mathtt{a}'\mathbb{S}) \qquad \mathtt{a} \Leftrightarrow \mathtt{a}' \triangleq \forall \mathbb{S}.(\mathtt{a}\ \mathbb{S} \leftrightarrow \mathtt{a}'\mathbb{S})$$
$$\neg \mathtt{a} \triangleq \lambda \mathbb{S}.\, \neg \mathtt{a}\ \mathbb{S} \qquad \mathtt{a} \wedge \mathtt{a}' \triangleq \lambda \mathbb{S}.\, \mathtt{a}\ \mathbb{S} \wedge \mathtt{a}'\mathbb{S}$$
$$\mathtt{a} \vee \mathtt{a}' \triangleq \lambda \mathbb{S}.\, \mathtt{a}\ \mathbb{S} \vee \mathtt{a}'\mathbb{S} \qquad \mathtt{a} \to \mathtt{a}' \triangleq \lambda \mathbb{S}.\, \mathtt{a}\ \mathbb{S} \to \mathtt{a}'\mathbb{S}$$
$$\forall x.\, \mathtt{a} \triangleq \lambda \mathbb{S}.\, \forall x.(\mathtt{a}\ \mathbb{S}) \qquad \exists x.\, \mathtt{a} \triangleq \lambda \mathbb{S}.\, \exists x.(\mathtt{a}\ \mathbb{S})$$
$$\mathtt{a} * \mathtt{a}' \triangleq \lambda(\mathbb{M}_0, \mathbb{R}).\, \exists \mathbb{M}, \mathbb{M}'.\, \mathbb{M}_0 = \mathbb{M} \uplus \mathbb{M}' \wedge \mathtt{a}(\mathbb{M}, \mathbb{R}) \wedge \mathtt{a}'(\mathbb{M}', \mathbb{R})$$

where $\mathbb{M} \uplus \mathbb{M}' \triangleq \mathbb{M} \cup \mathbb{M}'$ and $\mathsf{dom}(\mathbb{M}) \cap \mathsf{dom}(\mathbb{M}') = \emptyset$

Figure 11. Assertion operators

Specification language. Our specification language is defined in Fig 10. A code block $\mathbb{B}$ is a syntactic unit that represents a sequence $\mathbb{I}$ of instructions, beginning at specific memory address f. Note that in CAP, we usually insist that jump instructions can only appear at the end of a code block. This is no longer required in our new system so the division of code blocks is much more flexible.

The code heap $\mathbb{C}$ is a collection of code blocks that do not overlap, represented by a finite mapping from addresses to instruction sequences. Thus a code block can also be understood as a singleton code heap. To support Hoare-style reasoning, assertions are defined as predicates over GTM machine states (i.e., via "shallow embedding"). A program specification Ψ is a partial function which maps a memory address to its corresponding assertion, with the intention to represent the precondition of each code block. Thus, Ψ only has entries at each location that indicates the beginning of a code block.

Fig 11 defines an implication relation and a equivalence relation between two assertions ($\Rightarrow$) and also lifts all the standard logical operators to the assertion level. Note the difference between $\mathtt{a} \to \mathtt{a}'$

$$\mathtt{blk}(\mathtt{f}:\mathbb{I}) \triangleq \begin{cases} \lambda\mathbb{S}.\mathtt{Decode}(\mathbb{S},\mathtt{f},\iota) & \text{if } \mathbb{I}=\iota \\ \lambda\mathbb{S}.\mathtt{Decode}(\mathbb{S},\mathtt{f},\iota)\wedge(\mathtt{blk}(\mathtt{f}+|\mathsf{Ec}(\iota)|:\mathbb{I}')\ \mathbb{S}) & \text{if } \mathbb{I}=\iota;\mathbb{I}' \end{cases}$$

$$\mathtt{blk}(\mathbb{C}) \triangleq \forall \mathtt{f}\in\mathsf{dom}(\mathbb{C}).\mathtt{blk}(\mathtt{f}:\mathbb{C}(\mathtt{f}))$$

$$(\Psi_1\cup\Psi_2)(\mathtt{f}) \triangleq \begin{cases} \Psi_1(\mathtt{f}) & \text{if } \mathtt{f}\in\mathsf{dom}(\Psi_1)\setminus\mathsf{dom}(\Psi_2) \\ \Psi_2(\mathtt{f}) & \text{if } \mathtt{f}\in\mathsf{dom}(\Psi_2)\setminus\mathsf{dom}(\Psi_1) \\ \Psi_1(\mathtt{f})\vee\Psi_2(f) & \text{if } \mathtt{f}\in\mathsf{dom}(\Psi_1)\cap\mathsf{dom}(\Psi_2) \end{cases}$$

Figure 12. Predefined functions

$\boxed{\Psi\vdash\mathbb{W}}$ (***Well-formed World***)

$$\frac{\Psi\vdash\mathbb{C}:\Psi \quad (\mathtt{a}*(\mathtt{blk}(\mathbb{C})\wedge\mathtt{blk}(\mathtt{pc}:\mathbb{I}))\ \mathbb{S} \quad \Psi\vdash\{\mathtt{a}\}\mathtt{pc}:\mathbb{I}}{\Psi\vdash(\mathbb{S},\mathtt{pc})}\ (\textsc{prog})$$

$\boxed{\Psi\vdash\mathbb{C}:\Psi'}$ (***Well-formed Code Heap***)

$$\frac{\Psi_1\vdash\mathbb{C}_1:\Psi_1' \quad \Psi_2\vdash\mathbb{C}_2:\Psi_2' \quad \mathsf{dom}(\mathbb{C}_1)\cap\mathsf{dom}(\mathbb{C}_2)=\emptyset}{\Psi_1\cup\Psi_2\vdash\mathbb{C}_1\cup\mathbb{C}_2:\Psi_1'\cup\Psi_2'}\ (\textsc{link-c})$$

$$\frac{\Psi\vdash\{\mathtt{a}\}\mathtt{f}:\mathbb{I}}{\Psi\vdash\{\mathtt{f}\rightsquigarrow\mathbb{I}\}:\{\mathtt{f}\rightsquigarrow\mathtt{a}\}}\ (\textsc{cdhp})$$

$\boxed{\Psi\vdash\{\mathtt{a}\}\mathbb{B}}$ (***Well-formed Code Block***)

$$\frac{\Psi\vdash\{\mathtt{a}'\}(\mathtt{f}+|\mathsf{Ec}(\iota)|):\mathbb{I} \qquad \Psi\cup\{\mathtt{f}+|\mathsf{Ec}(\iota)|\rightsquigarrow\mathtt{a}'\}\vdash\{\mathtt{a}\}\mathtt{f}:\iota}{\Psi\vdash\{\mathtt{a}\}\mathtt{f}:\iota;\mathbb{I}}\ (\textsc{seq})$$

$$\frac{\forall\mathbb{S}.\mathtt{a}\ \mathbb{S}\rightarrow\Psi(\mathsf{Npc}_{\mathtt{f},\iota}(\mathbb{S}))\ (\mathsf{Next}_{\mathtt{f},\iota}(\mathbb{S}))}{\Psi\vdash\{\mathtt{a}\}\mathtt{f}:\iota}\ (\textsc{instr})$$

Figure 13. Inference rules for GCAP0

and $\mathtt{a}\Rightarrow\mathtt{a}'$: the former is an assertion, while the latter is a proposition! We also define standard separation logic primitives [14, 28] as assertion operators. The separating conjunction ($*$) of two assertions holds if they can be satisfied on two separating memory areas (the register file can be shared). Separating implication, empty heap, or singleton heap can also be defined directly in our meta logic.

Fig 12 defines a few important macros: $\mathtt{blk}(\mathbb{B})$ holds if $\mathbb{B}$ is stored properly in the memory of the current state; $\mathtt{blk}(\mathbb{C})$ holds if all code blocks in the code heap $\mathbb{C}$ are properly stored. The union of two program specifications is just the disjunction of the two corresponding assertions at each address.

Inference rules. Fig 13 presents the inference rules of GCAP0. We give three sets of judgments (from local to global): well-formed code block, well-formed code heap, and well-formed world.

Intuitively, a code block is well-formed ($\Psi\vdash\{\mathtt{a}\}\mathbb{B}$) iff, starting from a state satisfying its precondition $\mathtt{a}$, the code block is safe to execute until it jumps to a location in a state satisfying the specification Ψ. The well-formedness of a single instruction (rule INSTR) directly follows this understanding. Inductively, to validate the well-formedness of a code block beginning with ι under precondition $\mathtt{a}$ (rule SEQ), we should find an intermediate assertion $\mathtt{a}'$ serving simultaneously as the precondition of the tail code sequence, and the postcondition of ι. In the second premise of SEQ, since our syntax does not have a postcondition, $\mathtt{a}'$ is directly fed into the accompanied specification.

Note that for a well-formed code block, even though we have added an extra entry to the program specification Ψ when we validate each individual instruction, the Ψ used for validating each code block and the tail code sequence remains the same.

We can instantiate the INSTR and SEQ rules on each instruction if necessary. For example, specializing INSTR over the direct jump ($\mathtt{j}\ \mathtt{f}'$) results in the following rule:

$$\frac{\mathtt{a}\Rightarrow\Psi(\mathtt{f}')}{\Psi\vdash\{\mathtt{a}\}\mathtt{f}:\mathtt{j}\ \mathtt{f}'}\ (\textsc{j})$$

Specializing SEQ over the add instruction makes

$$\frac{\Psi\vdash\{\mathtt{a}'\}\mathtt{f}+4:\mathbb{I} \qquad \Psi\cup\{\mathtt{f}+4\rightsquigarrow\mathtt{a}'\}\vdash\{\mathtt{a}\}\mathtt{f}:\mathsf{add}\ \mathtt{r}_d,\mathtt{r}_s,\mathtt{r}_t}{\Psi\vdash\{\mathtt{a}\}\mathtt{f}:\mathsf{add}\ \mathtt{r}_d,\mathtt{r}_s,\mathtt{r}_t;\mathbb{I}}$$

which via INSTR can be further reduced into

$$\frac{\begin{array}{l}\Psi\vdash\{\mathtt{a}'\}\mathtt{f}+4:\mathbb{I}\\ \forall(\mathbb{M},\mathbb{R}).\ \mathtt{a}\ (\mathbb{M},\mathbb{R})\rightarrow\mathtt{a}'\ (\mathbb{M},\mathbb{R}\{\mathtt{r}_d\rightsquigarrow\mathbb{R}(\mathtt{r}_s)+\mathbb{R}(\mathtt{r}_t)\})\end{array}}{\Psi\vdash\{\mathtt{a}\}\mathtt{f}:\mathsf{add}\ \mathtt{r}_d,\mathtt{r}_s,\mathtt{r}_t;\mathbb{I}}\ (\textsc{add})$$

Another interesting case is the conditional jump instructions, such as beq, which can be instantiated from rule SEQ as

$$\frac{\begin{array}{l}\Psi\vdash\{\mathtt{a}'\}(\mathtt{f}+4):\mathbb{I} \quad \forall(\mathbb{M},\mathbb{R}).\ \mathtt{a}\ (\mathbb{M},\mathbb{R})\rightarrow((\mathbb{R}(\mathtt{r}_s)=\mathbb{R}(\mathtt{r}_t)\rightarrow\\ \quad\Psi(\mathtt{f}+i)\ (\mathbb{M},\mathbb{R}))\wedge(\mathbb{R}(\mathtt{r}_s)\neq\mathbb{R}(\mathtt{r}_t)\rightarrow\mathtt{a}'\ (\mathbb{M},\mathbb{R})))\end{array}}{\Psi\vdash\{\mathtt{a}\}\mathtt{f}:\mathsf{beq}\ \mathtt{r}_s,\mathtt{r}_t,i;\mathbb{I}}\ (\textsc{beq})$$

The instantiated rules are straightforward to understand and convenient to use. Most importantly, they can be automatically generated directly from the operational semantics for GTM.

The well-formedness of a code heap ($\Psi\vdash\mathbb{C}:\Psi'$) states that given Ψ' specifying the preconditions of each code block of $\mathbb{C}$, all the code in $\mathbb{C}$ can be safely executed with respect to specification Ψ. Here the domain of $\mathbb{C}$ and Ψ' should always be the same. The CDHP rule casts a code block into a corresponding well-formed singleton code heap, and the LINK-C rule merges two disjoint well-formed code heaps into a larger one.

A world is well-formed with respect to a global specification Ψ (the PROG rule), if

- the entire code heap is well-formed with respect to Ψ;
- the code heap and the current code block is properly stored;
- A precondition $\mathtt{a}$ is satisfied, separately from the code section;
- the instruction sequence is well-formed under $\mathtt{a}$.

The PROG rule also shows how we use separation conjunction to ensure that the whole code heap is indeed in the memory and always immutable; because assertion $\mathtt{a}$ cannot refer to the memory region occupied by $\mathbb{C}$, and the memory domain never grow during the execution of a program, the whole reasoning process below the top level never involves the code heap region. This guarantees that no code-modification can happen during the program execution.

To verify the safety and correctness of a program, one needs to first establish the well-formedness of each code block. All the code blocks are linked together progressively, resulting in a well-formed global code heap where the two accompanied specifications must match. Finally, the PROG rule is used to prove the safety of the initial world for the program.

Soundness and frame rules. The soundness of GCAP0 guarantees that any well-formed world is safe to execute. Establishing a well-formed world is equivalent to an invariant-based proof of program correctness: the accompanied specification Ψ corresponds to a global invariant that the current world satisfies.

Theorem 3.3 (Soundness of GCAP0) If $\Psi\vdash\mathbb{W}$, then $\mathsf{Safe}(\mathbb{W})$.

Detailed formal proofs can be found in our TR [5]. The following lemma (a.k.a., the frame rule) captures the essence of local reasoning for separation logic:

Lemma 3.4 $$\frac{\Psi\vdash\{\mathtt{a}\}\mathbb{B}}{(\lambda\mathtt{f}.\Psi(\mathtt{f})*\mathtt{a}')\vdash\{\mathtt{a}*\mathtt{a}'\}\mathbb{B}}\ (\textsc{frame-block})$$

where $\mathtt{a}'$ is independent of every register modified by $\mathbb{B}$.

$\boxed{\Psi \vdash \mathbb{W}}$ (*Well-formed World*)

$$\frac{\Psi \vdash (\mathbb{C}, \Psi) \quad \mathbb{C}' \subseteq \mathbb{C} \quad (\mathtt{a} * (\mathtt{blk}(\mathbb{C}') \wedge \mathtt{blk}(\mathtt{pc} : \mathbb{I}))) \; \mathbb{S} \quad \Psi' \vdash \{\mathtt{a}\}\mathtt{pc} : \mathbb{I} \quad \forall \mathtt{f} \in \mathsf{dom}(\Psi').(\Psi'(\mathtt{f}) * (\mathtt{blk}(\mathbb{C}') \wedge \mathtt{blk}(\mathtt{pc} : \mathbb{I})) \Rightarrow \Psi(\mathtt{f}))}{\Psi \vdash (\mathbb{S}, \mathtt{pc})} \text{ (PROG-G)}$$

$\boxed{\Psi \vdash (\mathbb{C}, \Psi')}$ (*Well-formed Code Specification*)

$$\frac{\Psi_1 \vdash (\mathbb{C}_1, \Psi_1') \quad \Psi_2 \vdash (\mathbb{C}_2, \Psi_2') \quad \mathsf{dom}(\mathbb{C}_1) \cap \mathsf{dom}(\mathbb{C}_2) = \emptyset}{\Psi_1 \cup \Psi_2 \vdash (\mathbb{C}_1 \cup \mathbb{C}_2, \Psi_1' \cup \Psi_2')} \text{ (LINK-G)}$$

$$\frac{\Psi \vdash \mathbb{C} : \Psi'}{\Psi * \mathtt{blk}(\mathbb{C}) \vdash (\mathbb{C}, \Psi' * \mathtt{blk}(\mathbb{C}))} \text{ (LIFT)}$$

Figure 14. Inference rules for GCAP1

Note that the correctness of this rule relies on the condition we gave in Sec 2 (incorporating extra memory does not affect the program execution), as also pointed out by Reynolds [28].

With the FRAME-BLOCK rule, one can extend a locally certified code block with an extra assertion, given the requirement that this assertion holds separately in conjunction with the original assertion as well as the specification. Frame rules at different levels will be used as the main tool to divide code and data to solve the SMC issue later. All the derived rules and the soundness proof have been fully mechanized in Coq [5] and will be used freely in our examples.

4. Certifying Runtime Code Generation

GCAP1 is a simple extension of GCAP0 to support runtime code generation. In the top PROG rule for GCAP0, the precondition a for the current code block must not specify memory regions occupied by the code heap, and all the code must be stored in the memory and remain immutable during the whole execution process. In the case of runtime code generation, this requirement has to be relaxed since the entire code may not be in the memory at the very beginning—some can be generated dynamically!

Inference rules. GCAP1 borrows the same definition of well-formed code heaps and well-formed code blocks as in GCAP0: they use the same set of inference rules (see Fig 13). To support runtime code generation, we change the top rule and insert an extra layer of judgments called well-formed code specification (see Fig 14) between well-formed world and well-formed code heap.

If "well-formed code heap" is a static reasoning layer, "well-formed code specification" is more like a dynamic one. Inside an assertion for a well-formed code heap, no information about program code is included, since it is implicitly guaranteed by the code immutability property. For a well-formed code specification, on the other hand, all information about the required program code should be provided in the precondition for all code blocks.

We use the LIFT rule to transform a well-formed code heap into a well-formed code specification by attaching the whole code information to the specifications on both sides. LINK-G rule has the same form as LINK-C, except that it works on the dynamic layer.

The new top rule (PROG-G) replaces a well-formed code heap with a well-formed code specification. The initial condition is now weakened! Only the current (locally immutable) code heap with the current code block, rather than the whole code heap, is required to be in the memory. Also, when proving the well-formedness of the current code block, the current code heap information is stripped from the global program specification.

Local reasoning. On the dynamic reasoning layer, since code information is carried with assertions and passed between code modules all the time, verification of one module usually involves the knowledge of code of another (as precondition). Sometimes, such knowledge is redundant and breaks local reasoning. Fortunately, a frame rule can be established on the code specification level as well. We can first locally verify the module, then extend it with the frame rule so that it can be linked with other modules later.

The original code:

```
B1 { main:  la   $9, gen          # get the target addr
     li   $8, 0xac880000  # load Ec(sw $8,0($4))
     sw   $8, 0($9)       # store to gen
     li   $8, 0x00800008  # load Ec(jr $4)
     sw   $8, 4($9)       # store to gen+4
     la   $4, ggen        # $4 = ggen
     la   $9, main        # $9 = main
     li   $8, 0x01200008  # load Ec(jr $9) to $8
     j    gen             # jump to target
   gen:  nop               # to be generated
     nop                  # to be generated
   ggen: nop               # to be generated
```

The generated code:

```
B2 { gen:  sw   $8,0($4)
     jr   $4

B3 { ggen: jr   $9
```

Figure 15. `mrcg.s`: Multilevel runtime code generation

Lemma 4.1 $$\frac{\Psi \vdash (\mathbb{C}, \Psi')}{(\lambda \mathtt{f}.\Psi(\mathtt{f}) * \mathtt{a}) \vdash (\mathbb{C}, \lambda \mathtt{f}.\Psi'(\mathtt{f}) * \mathtt{a})} \text{ (FRAME-SPEC)}$$

where a is independent of any register that is modified by $\mathbb{C}$.

Proof: By induction over the derivation for $\Psi \vdash (\mathbb{C}, \Psi')$. There are only two cases: if the final step is done via the LINK-G rule, the conclusion follows immediately from the induction hypothesis; if the final step is via the LIFT rule, it must be derived from a well-formed-code-heap derivation:

$$\Psi_0 \vdash \mathbb{C} : \Psi_0' \tag{1}$$

with $\Psi = \lambda \mathtt{f}.\Psi_0(\mathtt{f}) * \mathtt{blk}(\mathbb{C})$ and $\Psi' = \lambda \mathtt{f}.\Psi_0'(\mathtt{f}) * \mathtt{blk}(\mathbb{C})$; we first apply the FRAME-CDHP rule to (1) obtain:

$$(\lambda \mathtt{f}.\Psi_0(\mathtt{f}) * \mathtt{a}) \vdash \mathbb{C} : \lambda \mathtt{f}.\Psi_0'(\mathtt{f}) * \mathtt{a}$$

and then apply the LIFT rule to get the conclusion. ∎

In particular, by setting the above assertion a to be the knowledge about code not touched by the current module, the code can be excluded from the local verification.

As a more concrete example, suppose that we have two locally certified code modules $\mathbb{C}_1$ and $\mathbb{C}_2$, where $\mathbb{C}_2$ is generated by $\mathbb{C}_1$ at runtime. We first apply FRAME-SPEC to extend $\mathbb{C}_2$ with assertion $\mathtt{blk}(\mathbb{C}_1)$, which reveals the fact that $\mathbb{C}_1$ does not change during the whole executing process of $\mathbb{C}_2$. After this, the LINK-G rule is applied to link them together into a well-formed code specification. We give more examples about GCAP1 in Section 6.

Soundness. The soundness of GCAP1 can be established in the same way as Theorem 3.3 (see TR [5] for more detail).

Theorem 4.2 (Soundness of GCAP1) If $\Psi \vdash \mathbb{W}$, then $\mathsf{Safe}(\mathbb{W})$.

To verify a program that involves run-time code generation, we first establish the well-formedness of each code module (which never modifies its own code) using the rules for well-formed code heap as in GCAP0. We then use the dynamic layer to combine these code modules together into a global code specification. Finally we use the new PROG-G rule to establish the initial state and prove the correctness of the entire program.

Example: Multilevel Runtime Code Generation We use a small example `mrcg.s` in Fig 15 to demonstrate the usability of GCAP1 on runtime code generation. Our `mrcg.s` is already fairly subtle—it does multilevel RCG, which means that code generated at runtime may itself generate new code. Multilevel RCG has its practical usage [13]. In this example, the code block $\mathbb{B}_1$ can generate $\mathbb{B}_2$ (containing two instructions), which will again generate $\mathbb{B}_3$ (containing only a single instruction).

The first step is to verify $\mathbb{B}_1$, $\mathbb{B}_2$ and $\mathbb{B}_3$ respectively and locally, as the following three judgments show:

$$\{\texttt{gen} \rightsquigarrow \mathtt{a}_2 * \texttt{blk}(\mathbb{B}_2)\} \vdash \{\mathtt{a}_1\}\mathbb{B}_1$$
$$\{\texttt{ggen} \rightsquigarrow \mathtt{a}_3 * \texttt{blk}(\mathbb{B}_3)\} \vdash \{\mathtt{a}_2\}\mathbb{B}_2$$
$$\{\texttt{main} \rightsquigarrow \mathtt{a}_1\} \vdash \{\mathtt{a}_3\}\mathbb{B}_3$$

where

$\mathtt{a}_1 = \lambda\mathbb{S}.\texttt{True}$,

$\mathtt{a}_2 = \lambda(\mathbb{M},\mathbb{R}).\mathbb{R}(\$9) = \texttt{main} \wedge \mathbb{R}(\$8) = \mathsf{Ec}(\mathsf{jr}\ \$9) \wedge \mathbb{R}(\$4) = \texttt{ggen}$,

$\mathtt{a}_3 = \lambda(\mathbb{M},\mathbb{R}).\mathbb{R}(\$9) = \texttt{main}$

As we see, $\mathbb{B}_1$ has no requirement for its precondition, $\mathbb{B}_2$ simply requires that proper values are stored in the registers \$4, \$8, and \$9, while $\mathbb{B}_3$ demands that \$9 points to the label `main`.

All the three judgments are straightforward to establish, by means of GCAP1 inference rules (the SEQ rule and the INSTR rule). For example, the pre- and selected intermediate conditions for $\mathbb{B}_1$ are as follows:

```
            {λ𝕊.True}
main:       la   $9, gen
            {λ(𝕄,ℝ).ℝ($9) = gen}
            li   $8, 0xac880000
            sw   $8, 0($9)
            {(λ(𝕄,ℝ).ℝ($9) = gen)∗blk(gen : sw $8, 0($4))}
            li   $8, 0x00800008
            sw   $8, 4($9)
            {blk(𝔹2)}
            la   $4, ggen
            la   $9, main
            li   $8, 0x01200008
            {a2 ∗ blk(𝔹2)}
            j    gen
```

The first five instructions generate the body of $\mathbb{B}_2$. Then, registers are stored with proper values to match $\mathbb{B}_2$'s requirement. Notice the three `li` instructions: the encoding for each generated instruction are directly specified as immediate value here.

Notice that $\texttt{blk}(\mathbb{B}_1)$ has to be satisfied as a precondition of $\mathbb{B}_3$ since $\mathbb{B}_3$ points to $\mathbb{B}_1$. However, to achieve modularity we do not require it in $\mathbb{B}_3$'s local precondition. Instead, we leave this condition to be added later via our frame rule.

After the three code blocks are locally certified, the CDHP rule and then the LIFT rule are respectively applied to each of them, as illustrated in Fig 16, resulting in three well-formed singleton code heaps. Afterwards, $\mathbb{B}_2$ and $\mathbb{B}_3$ are linked together and we apply FRAME-SPEC rule to the resulting code heap, so that it can successfully be linked together with the other code heap, forming the coherent global well-formed specification (as Fig 16 indicates):

$$\Psi_G = \{\texttt{main} \rightsquigarrow \mathtt{a}_1 * \texttt{blk}(\mathbb{B}_1),\ \texttt{gen} \rightsquigarrow \mathtt{a}_2 * \texttt{blk}(\mathbb{B}_2) * \texttt{blk}(\mathbb{B}_1),$$
$$\texttt{ggen} \rightsquigarrow \mathtt{a}_3 * \texttt{blk}(\mathbb{B}_3) * \texttt{blk}(\mathbb{B}_1)\}$$

which should satisfy $\Psi_G \vdash (\mathbb{C}, \Psi_G)$ (where $\mathbb{C}$ stands for the entire code heap).

Now we can finish the last step—applying the PROG-G rule to the initial world, so that the safety of the whole program is assured.

5. Supporting General SMC

Although GCAP1 is a nice extension to GCAP0, it can hardly be used to certify general SMC. For example, it cannot verify the opcode modification example given in Fig 8 at the end of Sec 2.

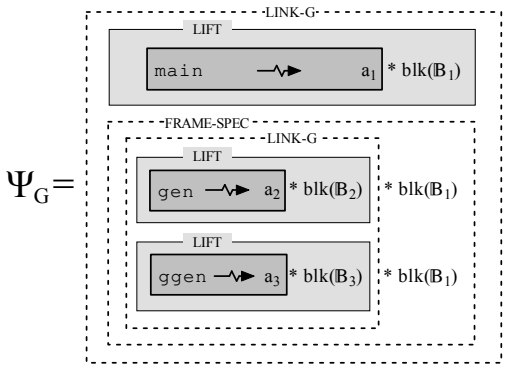

Figure 16. `mrcg.s`: GCAP1 specification

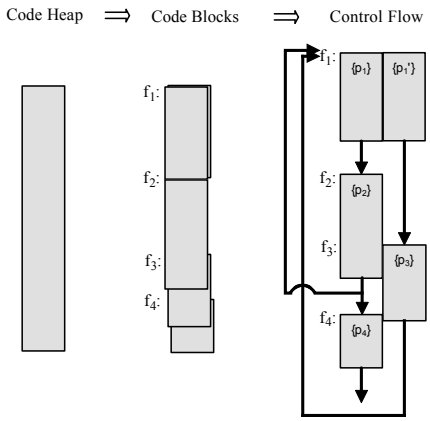

Figure 17. Typical Control Flow in GCAP2

$$(\textit{ProgSpec}) \quad \Psi \in \textit{Address} \rightharpoonup \textit{Assertion}$$
$$(\textit{CodeSpec}) \quad \Phi \in \textit{CodeBlock} \rightharpoonup \textit{Assertion}$$

Figure 18. Assertion language for GCAP2

In fact, GCAP1 will not even allow the same memory region to contain different runtime instructions.

General SMC does not distinguish between code heap and data heap, therefore poses new challenges: first, at runtime, the instructions stored at the same memory location may vary from time to time; second, the control flow is much harder to understand and represent; third, it is unclear how to divide a self-modifying program into code blocks so that they can be reasoned about separately.

To tackle these obstacles, we have developed a new verification system GCAP2 supporting general SMCs. Our system is still built upon our machine model GTM.

The main idea of GCAP2 is illustrated in Fig 17. Again, the potential runtime code is decomposed into code blocks, representing the instruction sequences that may possibly be executed. Each code block is assigned with a precondition, so that it can be certified individually. Unlike GCAP1, since instructions can be modified, different runtime code blocks may overlap in memory, even share the same entry location. Hence if a code block contains a jump instruction to certain memory address (such as to $\mathtt{f}_1$ in Fig 17) at which several blocks start, it is usually not possible to tell statically which block it will exactly jump to at runtime. What our system requires instead is that whenever the program counter reaches this address (e.g. $\mathtt{f}_1$ in Fig 17), there should *exist* at least one code block there, whose precondition is matched. After all the code blocks are cer-

$\boxed{\Psi \vdash \mathbb{W}}$ ***(Well-formed World)***

$$\frac{[\![\Phi]\!] \vdash \Phi \quad \mathtt{a}\ \mathbb{S} \quad [\![\Phi]\!] \vdash \{\mathtt{a}\}\mathtt{pc}:\mathbb{I}}{[\![\Phi]\!] \vdash (\mathbb{S}, \mathtt{pc})}\ (\textsc{prog})$$

where $[\![\Phi]\!] \triangleq \lambda \mathtt{f}.\, \exists \mathbb{I}.\, \Phi(\mathtt{f}:\mathbb{I})$.

$\boxed{\Psi \vdash \Phi}$ ***(Well-formed Code Specification)***

$$\frac{\Psi_1 \vdash \Phi_1 \qquad \Psi_2 \vdash \Phi_2 \qquad \mathsf{dom}(\Phi_1) \cap \mathsf{dom}(\Phi_2) = \emptyset}{\Psi_1 \cup \Psi_2 \vdash \Phi_1 \cup \Phi_2}\ (\textsc{link})$$

$$\frac{\forall \mathbb{B} \in \mathsf{dom}(\Phi).\, \Psi \vdash \{\Phi\ \mathbb{B}\}\mathbb{B}}{\Psi \vdash \Phi}\ (\textsc{cdhp})$$

$\boxed{\Psi \vdash \{\mathtt{a}\}\mathbb{B}}$ ***(Well-formed Code Block)***

$$\frac{\Psi \vdash \{\mathtt{a}'\}\mathtt{f}+|\mathsf{Ec}(\iota)|:\mathbb{I} \qquad \Psi \cup \{\mathtt{f}+|\mathsf{Ec}(\iota)| \rightsquigarrow \mathtt{a}'\} \vdash \{\mathtt{a}\}\mathtt{f}:\iota}{\Psi \vdash \{\mathtt{a}\}\mathtt{f}:\iota;\mathbb{I}}\ (\textsc{seq})$$

$$\frac{\forall \mathbb{S}.\, \mathtt{a}\ \mathbb{S} \rightarrow (\ \mathtt{Decode}(\mathbb{S}, \mathtt{f}, \iota) \wedge \Psi(\mathsf{Npc}_{\mathtt{f},\iota}(\mathbb{S}))\ \mathsf{Next}_{\mathtt{f},\iota}(\mathbb{S}))}{\Psi \vdash \{\mathtt{a}\}\mathtt{f}:\iota}\ (\textsc{instr})$$

Figure 19. Inference rules for GCAP2

tified, they can be linked together in a certain way to establish the correctness of the program.

To support self-modifying features, we relax the requirements of well-formed code blocks. Specifically, a well-formed code block now describes an *execution sequence* of instructions starting at certain memory address, rather than merely a static instruction sequence currently stored in memory. There is no difference between these two understandings under the non-self-modifying circumstance since the static code always executes as it is, while a fundamental difference could appear under the more general SMC cases. The new understanding *execution code block* characterizes better the real control flow of the program. Our TR discusses more about the importance of this generalization.

Specification language. The specification language is almost same as GCAP1, but GCAP2 introduces one new concept called *code specification* (denoted as Φ in Fig 18), which generalizes the previous code and specification pair to resolve the problem of having multiple code blocks starting at a single address. A code specification is a partial function that maps code blocks to their assertions. When certifying a program, the domain of the global Φ indicates all the code blocks that can show up at runtime, and the corresponding assertion of a code block describes its global precondition. The reader should note that though Φ is a partial function, it can have an infinite domain (indicating that there might be an infinite number of possible runtime code blocks).

Inference rules. GCAP2 has three sets of judgements (see Fig 19): well-formed world, well-formed code spec, and well-formed code block. The key idea of GCAP2 is to eliminate the well-formed-code-heap layer in GCAP1 and push the "dynamic reasoning layer" down inside each code block, even into a single instruction. Interestingly, this makes the rule set of GCAP2 look much like GCAP0 rather than GCAP1.

The inference rules for well-formed code blocks has one tiny but essential difference from GCAP0/GCAP1. A well-formed instruction (INSTR) has one more requirement that the instruction must actually be in the proper location of memory. Previously in GCAP1, this is guaranteed by the LIFT rule which adds the whole static code heap into the preconditions; for GCAP2, it is only required that the current executing instruction be present in memory.

Intuitively, the well-formedness of a code block $\Psi \vdash \{\mathtt{a}\}\mathtt{f}:\mathbb{I}$ now states that if a machine state satisfies assertion $\mathtt{a}$, then $\mathbb{I}$ is the only possible code sequence to be executed starting from $\mathtt{f}$, until we get to a program point where the specification Ψ can be matched.

The precondition for a non-self-modifying code block $\mathbb{B}$ must now include $\mathbb{B}$ itself, i.e. $\mathtt{blk}(\mathbb{B})$. This extra requirement does not compromise modularity, since the code is already present and can be easily moved into the precondition. For dynamic code, the initial stored code may differ from the code actually being executed.

Note that our generalization does not make the verification more difficult: as long as the specification and precondition are given, the well-formedness of a code block can be established in the same mechanized way as before.

The judgment $\Psi \vdash \Phi$ (well-formed code specification) is fairly comparable with the corresponding judgment in GCAP1 if we notice that the pair $(\mathbb{C}, \Psi)$ is just a way to represent a more limited Φ. The rules here basically follow the same idea except that the CDHP rule allows universal quantification over code blocks: if every block in a code specification's domain is well-formed with respect to a program specification, then the code specification is well-formed with respect to the same program specification.

The interpretation operator $[\![-]\!]$ establishes the semantic relation between program specifications and code specifications: it transforms a code specification to a program specification by uniting the assertions (i.e. doing assertion disjunction) of all blocks starting at the same address together. In the judgment for well-formed world (rule PROG), we use $[\![\Phi]\!]$ as the specification to establish the well-formed code specification and the current well-formed code block. We do not need to require the current code block to be stored in memory (as GCAP1 did) since such requirement will be specified in the assertion $\mathtt{a}$ already.

Soundness and local reasoning. The soundness proof follows almost the same techniques as in GCAP0.

Theorem 5.1 (Soundness of GCAP2) If $\Psi \vdash \mathbb{W}$, then $\mathsf{Safe}(\mathbb{W})$.

Frame rules are still the key idea for supporting local reasoning. In fact, since we no longer have the static code layer in GCAP2, the frame rules play a more important role in achieving modularity. For example, to link two code modules that do not modify each other, we first use the frame rule to feed the code information of the other module into each module's specification and then apply LINK rule.

Example: Opcode modification. We can now use GCAP2 to certify the opcode-modification example given in Fig 8. There are four runtime code blocks that need to be handled. Fig 20 shows the formal specification for each code block, including both the local version and the global version. Note that $\mathbb{B}_1$ and $\mathbb{B}_2$ are overlapping in memory, so we cannot just use GCAP1 to certify this example.

Locally, we need to make sure that each code block is indeed stored in memory before it can be executed. To execute $\mathbb{B}_1$ and $\mathbb{B}_4$, we also require that the memory location at the address `new` stores a proper instruction (which will be loaded later). On the other hand, since $\mathbb{B}_4$ and $\mathbb{B}_2$ can be executed if and only if the branch occurs at `main`, they both have the precondition $\mathbb{R}(\$2) = \mathbb{R}(\$4)$.

After verifying all the code blocks based on their local specifications, we can apply the frame rule to establish the extended specifications. As Fig 20 shows, the frame rule is applied to the local judgments of $\mathbb{B}_2$ and $\mathbb{B}_4$, adding $\mathtt{blk}(\mathbb{B}_3)$ on their both sides to form the corresponding global judgments. And for $\mathbb{B}_1$, $\mathtt{blk}(\mathbb{B}_3) * \mathtt{blk}(\mathbb{B}_4)$ is added; here the additional $\mathtt{blk}(\mathbb{B}_4)$ in the specification entry for `halt` will be weakened out by the LINK rule (the union of two program specifications used in the LINK rule is defined in Fig 12).

Finally, all these judgments are joined together via the LINK rule to establish the well-formedness of the global code. This is similar to how we certify code using GCAP1 in the previous section, except that the LIFT process is no longer required here. The global code specification is exactly:

```
B1 { main:   beq  $2, $4, modify
     main:   move $2, $4          # B2 starts here
     main:   j    halt
B2 { target: addi $2, $2, 1
     target: j    halt
B3 { halt:   j    halt
B4 { modify: lw   $9, new
     modify: sw   $9, target
     modify: j    target
```

Let

$a_1 \triangleq \mathtt{blk}(\mathtt{new} : \mathsf{addi}\ \$2, \$2, 1) * \mathtt{blk}(\mathbb{B}_1)$

$a_1' \triangleq \mathtt{blk}(\mathbb{B}_3) * \mathtt{blk}(\mathbb{B}_4)$

$a_2 \triangleq (\lambda(\mathbb{M}, \mathbb{R}).\, \mathbb{R}(\$2) = \mathbb{R}(\$4)) * \mathtt{blk}(\mathbb{B}_2)$

$a_3 \triangleq (\lambda(\mathbb{M}, \mathbb{R}).\, \mathbb{R}(\$4) \le \mathbb{R}(\$2) \le \mathbb{R}(\$4) + 1)$

$a_4 \triangleq (\lambda(\mathbb{M}, \mathbb{R}).\, \mathbb{R}(\$2) = \mathbb{R}(\$4)) * \mathtt{blk}(\mathtt{new} : \mathsf{addi}\ \$2, \$2, 1) * \mathtt{blk}(\mathbb{B}_1)$

Then local judgments are as follows

$$\{\mathtt{modify} \rightsquigarrow a_4, \mathtt{halt} \rightsquigarrow a_3\} \vdash \{a_1\}\mathbb{B}_1$$
$$\{\mathtt{halt} \rightsquigarrow a_3\} \vdash \{a_2\}\mathbb{B}_2$$
$$\{\mathtt{halt} \rightsquigarrow a_3 * \mathtt{blk}(\mathbb{B}_3)\} \vdash \{a_3 * \mathtt{blk}(\mathbb{B}_3)\}\mathbb{B}_3$$
$$\{\mathtt{target} \rightsquigarrow a_2\} \vdash \{a_4 * \mathtt{blk}(\mathbb{B}_4)\}\mathbb{B}_4$$

After applying frame rules and linking, the global specification becomes

$$\{\mathtt{modify} \rightsquigarrow a_4 * a_1', \mathtt{halt} \rightsquigarrow a_3 * a_1'\} \vdash \{a_1 * a_1'\}\mathbb{B}_1$$
$$\{\mathtt{halt} \rightsquigarrow a_3 * \mathtt{blk}(\mathbb{B}_3)\} \vdash \{a_2 * \mathtt{blk}(\mathbb{B}_3)\}\mathbb{B}_2$$
$$\{\mathtt{halt} \rightsquigarrow a_3 * \mathtt{blk}(\mathbb{B}_3)\} \vdash \{a_3 * \mathtt{blk}(\mathbb{B}_3)\}\mathbb{B}_3$$
$$\{\mathtt{target} \rightsquigarrow a_2 * \mathtt{blk}(\mathbb{B}_3)\} \vdash \{a_4 * a_1'\}\mathbb{B}_4$$

Figure 20. `opcode.s`: Code and specification

$$\Phi_G = \{\mathbb{B}_1 \rightsquigarrow a_1 * a_1', \mathbb{B}_2 \rightsquigarrow a_2 * \mathtt{blk}(\mathbb{B}_3), \mathbb{B}_3 \rightsquigarrow a_3 * \mathtt{blk}(\mathbb{B}_3), \mathbb{B}_4 \rightsquigarrow a_4 * a_1'\}$$

which satisfies $[\![\Phi_G]\!] \vdash \Phi_G$ and ultimately the PROG rule can be successfully applied to validate the correctness of `opcode.s`. Actually we have proved not only the type safety of the program but also its partial correctness, for instance, whenever the program executes to the line `halt`, the assertion a_3 will always hold.

Parametric code. In SMC, as mentioned earlier, the number of code blocks we need to certify might be infinite. Thus, it is impossible to enumerate and verify them one by one. To resolve this issue, we introduce auxiliary variable(s) (i.e. parameters) into the code body, developing parametric code blocks and, correspondingly, parametric code specifications.

Traditional Hoare logic only allows auxiliary variables to appear in the pre- or post-condition of code sequences. In our new framework, by allowing parameters appearing in the code body and its assertion at the same time, assertions, code body and specifications can interact with each other. This make our program logic even more expressive.

One simplest case of parametric code block is as follows:

```
f:      li   $2, k
        j    halt
```

with the number k as a parameter. It simply represents a family of code blocks where k ranges over all possible natural numbers.

The code parameters can potentially be anything, e.g., instructions, code locations, or the operands of some instructions. Taking a whole code block or a code heap as parameter may allow us to express and prove more interesting applications.

Certifying parametric code makes use of the universal quantifier in the rule CDHP. In the example above we need to prove the judgment

$$\forall k.\, (\Psi_k \vdash \{\lambda\mathbb{S}.\, \mathtt{True}\}\mathtt{f} : \mathsf{li}\ \$2, k; \mathsf{j}\ \mathtt{halt})$$

$\{r_{DL} = 0x80 \wedge \mathbb{D}(512) = \mathsf{Ec}(\mathsf{jmp}\ -2) * \mathtt{blk}(\mathbb{B}_1)\}$

```
B1 { bootld:  movw   $0, %bx       # can not use ES
     bootld:  movw   %bx,%es       # kernel segment
     bootld:  movw   $0x1000,%bx   # kernel offset
     bootld:  movb   $1, %al       # num sectors
     bootld:  movb   $2, %ah       # disk read command
     bootld:  movb   $0, %ch       # starting cylinder
     bootld:  movb   $2, %cl       # starting sector
     bootld:  movb   $0, %dh       # starting head
     bootld:  int    $0x13         # call BIOS
     bootld:  ljmp   $0,$0x1000    # jump to kernel
```

$\{\mathtt{blk}(\mathbb{B}_2)\}$

```
B2 { kernel: jmp    $-2           # loop
```

Figure 21. `bootloader.s`: Code and specification

where $\Psi_k(\mathtt{halt}) = (\lambda(\mathbb{M}, \mathbb{R}).\, \mathbb{R}(\$2) = k)$, to guarantee that the parametric code block is well-formed with respect to the parametric specification Ψ.

Parametric code blocks are not just used in verifying SMC; they can be used in other circumstances. For example, to prove position independent code, i.e. code whose function does not depend on the absolute memory address where it is stored, we can parameterize the base address of that code to do the certification. Parametric code can also improve modularity, for example, by abstracting out certain code modules as parameters.

We will give more examples of parametric code blocks in Sec 6.

Expressiveness. The following important theorem shows the expressiveness of our GCAP2 system: as long as there exists an invariant for the safety of a program, GCAP2 can be used to certify it with a program specification which is equivalent to the invariant.

Theorem 5.2 (Expressiveness of GCAP2) If Inv is an invariant of GTM, then there is a Ψ, such that for any world $(\mathbb{S}, \mathtt{pc})$ we have

$$\mathsf{Inv}(\mathbb{S}, \mathtt{pc}) \longleftrightarrow ((\Psi \vdash (\mathbb{S}, \mathtt{pc})) \wedge (\Psi(\mathtt{pc})\ \mathbb{S})).$$

Together with the soundness theorem (Theorem 5.1), we have showed that there is a correspondence relation between a global program specification and a global invariant for any program.

It should also come as no surprise that any program certified under GCAP1 can always be translated into GCAP2. In fact, the judgments of GCAP1 and GCAP2 have very close connections. See our TR [5] for more discussions on both issues.

6. More Examples and Applications

We show the certification of a number of representative examples and applications using GCAP (see Table 1 in Sec 1). Due to the space limit, we can only give a few of these in this section. More examples can be found in our TR [5].

6.1 A Certified OS Boot Loader

An OS boot loader is a simple, yet prevalent application of runtime code loading. It is an artifact of a limited bootstrapping protocol, but one that continues to exist to this day. The limitation on an x86 architecture is that the BIOS of the machine will only load the first 512 bytes of code into main memory for execution. The boot loader is the code contained in those bytes that will load the rest of the OS from the disk into main memory, and begin executing the OS (Fig 22). Therefore certifying a boot loader is an important piece of a complete OS certification.

To show that we support a real boot loader, we have created one that runs on the Bochs simulator[18] and on a real x86 machine. The code (Fig 21) is very simple, it sets up the registers needed to

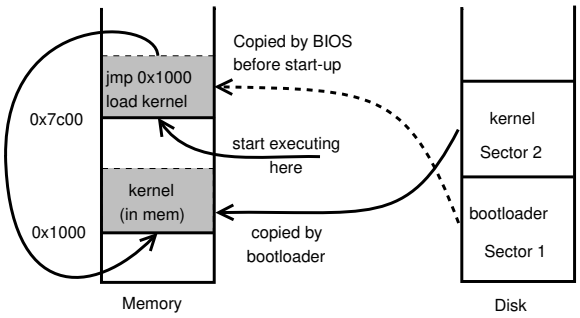

Figure 22. A typical boot loader

```
    .data   # Data declaration section

num:    .byte 8

    .text   # Code section

main:   lw   $4, num        # set argument
        lw   $9, key        # $9 = Ec(add $2,$2,0)
        li   $8, 1          # counter
        li   $2, 1          # accumulator

loop:   beq  $8, $4, halt   # check if done
        addi $8, $8, 1      # inc counter
        add  $10, $9, $2    # new instr to put
key:    addi $2, $2, 0      # accumulate
        sw   $10, key       # store new instr
        j    loop           # next round

halt:   j    halt
```

Figure 23. `fib.s`: Fibonacci number

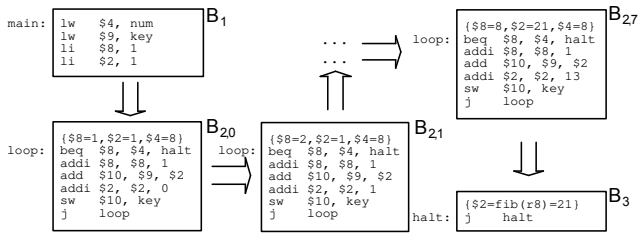

Figure 24. `fib.s`: Control flow

make a BIOS call to read the hard disk into the correct memory location, then makes the call to actually read the disk, then jumps to loaded memory.

The specifications of the boot loader are also simple. $\mathtt{r}_{DL} = 0x80$ makes sure that the number of the disk is given to the boot loader by the hardware. The value is passed unaltered to the int instruction, and is needed for that BIOS call to read from the correct disk. $\mathbb{D}(512) = \mathsf{Ec}(\mathsf{jmp}\ -2)$ makes sure that the disk actually contains a kernel with specific code. The code itself is not important, but the entry point into this code needs to be verifiable under a trivial precondition, namely that the kernel is loaded. The code itself can be changed. The boot loader proof will not change if the code changes, as it simply relies on the proof that the kernel code is certified. The assertion $\mathtt{blk}(\mathbb{B}_1)$ just says that boot loader is in memory when executed.

$\mathbb{B}_1$
```
{blk(B1)}
main:   lw   $4, num
        lw   $9, key
        li   $8, 1
        li   $2, 1
```
$\{(\lambda(\mathbb{M},\mathbb{R}).\mathbb{R}(\$4)=\mathbb{M}(\mathtt{num}) \wedge \mathbb{R}(\$9)=\mathsf{Ec}(\mathsf{addi}\ \$2,\$2,0) \wedge \mathbb{R}(\$8)=k{+}1 \wedge \mathbb{R}(\$2)=\mathsf{fib}(k{+}1)) * \mathtt{blk}(\mathbb{B}_{2,k})\}$

$\mathbb{B}_{2,k}$
```
loop:   beq  $8, $4, halt
        addi $8, $8, 1
        add  $10, $9, $2
key:    addi $2, $2, fib(k)
        sw   $10, key
        j    loop
```
$\{(\lambda(\mathbb{M},\mathbb{R}).\mathbb{R}(\$2)=\mathsf{fib}(\mathbb{M}(\mathtt{num}))) * \mathtt{blk}(\mathbb{B}_3)\}$

$\mathbb{B}_3$
```
halt:   j    halt
```

Figure 25. `fib.s`: Code and specification

6.2 Fibonacci Number and Parametric Code

To demonstrate the usage of parametric code, we construct an example `fib.s` to calculate the Fibonacci function $\mathsf{fib}(0) = 0$, $\mathsf{fib}(1) = 1$, $\mathsf{fib}(i{+}2) = \mathsf{fib}(i) + \mathsf{fib}(i{+}1)$, shown in Fig 23. More specifically, `fib.s` will calculate $\mathsf{fib}(\mathbb{M}(\mathtt{num}))$ which is $\mathsf{fib}(8) = 21$ and store it into register \$2.

It looks strange that this is possible since throughout the whole program, the only instructions that write \$2 is the fourth instruction which assigns 1 to it and the line `key` which does nothing.

The main trick, of course, comes from the code-modification instruction on the line next to `key`. In fact, the third operand of the addi instruction on the line `key` alters to the next Fibonacci number (temporarily calculated and stored in register \$10 before the instruction modification) during every loop. Fig 24 illustrates the complete execution process.

Since the opcode of the line `key` would have an unbounded number of runtime values, we need to seek help from parametric code blocks. The formal specifications for each code block is shown in Fig 25. We specify the program using three code blocks, where the second block—the kernel loop of our program $\mathbb{B}_{2,k}$—is a parametric one. The parameter k appears in the operand of the `key` instruction as an argument of the Fibonacci function.

Consider the execution of code block $\mathbb{B}_{2,k}$. Before it is executed, \$9 stores $\mathsf{Ec}(\mathsf{addi}\ \$2,\$2,0)$, and \$2 stores $\mathsf{fib}(k{+}1)$. Therefore, the execution of the third instruction addi \$10,\$9,\$2 changes \$10 into $\mathsf{Ec}(\mathsf{addi}\ \$2,\$2,\mathsf{fib}(k{+}1))$[1], so at the end of the loop, \$2 is now $\mathsf{fib}(k{+}1){+}\mathsf{fib}(k){=}\mathsf{fib}(k{+}2)$, and `key` has the instruction $\mathsf{addi}\ \$2,\$2,\mathsf{fib}(k{+}1)$ instead of $\mathsf{addi}\ \$2,\$2,\mathsf{fib}(k)$, then the program continues to the next loop $\mathbb{B}_{2,k+1}$.

The global code specification we finally get is as follows :

$$\Phi \triangleq \{\mathbb{B}_1 \rightsquigarrow \mathtt{a}_1,\ \mathbb{B}_{2k} \rightsquigarrow \mathtt{a}_{2k} \mid k{\in}\mathtt{Nat},\ \mathbb{B}_3 \rightsquigarrow \mathtt{a}_3\} \qquad (2)$$

But note that this formulation is just for readability; it is not directly expressible in our meta logic. To express parameterized code blocks, we need to use existential quantifiers. The example is in fact represented as $\Phi \triangleq \lambda\mathbb{B}.\lambda\mathbb{S}.(\mathbb{B}{=}\mathbb{B}_1 \wedge \mathtt{a}_1\ \mathbb{S}) \vee (\exists k.\mathbb{B}{=}\mathbb{B}_{2k} \wedge \mathtt{a}_{2k}\ \mathbb{S}) \vee (\mathbb{B}{=}\mathbb{B}_3 \wedge \mathtt{a}_3\ \mathbb{S})$. One can easily see the equivalence between this definition and (2).

6.3 Self Replication

Combining self-reading code with runtime code generation, we can produce self-growing program, which keeps replicating itself forever. This kind of code appears commonly in Core War—a game where different people write assembly programs that attack the other programs. Our demo code `selfgrow.s` is shown in Fig 26.

[1] To simplify the case, we assume the encoding of addi instruction has a linear relationship with respect to its numerical operand in this example.

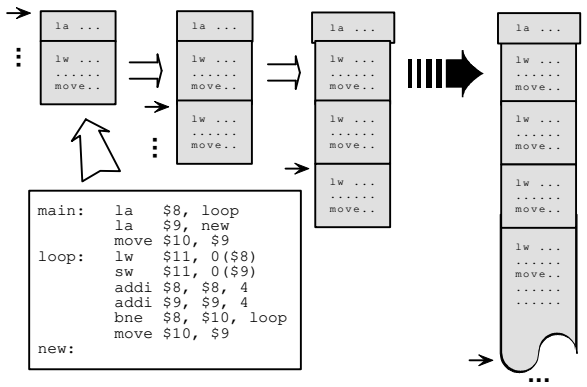

Figure 26. `selfgrow.s`: Self-growing process

$\mathbb{B}_0$ {blk($\mathbb{B}_0$)}

```
main:  la   $8, loop
       la   $9, new
       move $10, $9
```

{($\lambda(\mathbb{M},\mathbb{R}).\exists 0\le i<6.\mathbb{R}(\$8)=k+4i\wedge\mathbb{R}(\$9)=k+24+4i\wedge \mathbb{R}(\$10)=k+24\wedge\forall 0\le j<4i.\mathbb{M}(k+j)=\mathbb{M}(k+24+j)) * \mathtt{blk}(\mathbb{B}_k)$}

$\mathbb{B}_k$

```
k:     lw   $11, 0($8)
       sw   $11, 0($9)
       addi $8, $8, 4
       addi $9, $9, 4
       bne  $8, $10, k
       move $10, $9
```

Figure 27. `selfgrow.s`: Code and specification

After initializing the registers, the code repeatedly duplicates itself and continue to execute the new copy .

The block starting at `loop` is the code body that keeps being duplicated. During the execution, this part is copied to the `new` location. Then the program continues executing from `new`, until another code block is duplicated. Note that here we rely on the property that instruction encodings for branches use relative addressing, thus every time our code is duplicated, the target address of the `bne` instruction would change accordingly.

The copying process goes on and on, till the whole available memory is consumed and, presumably, the program would crash. However, under our assumption that the memory domain is infinite, this code never kill itself and thus can be certified.

The specification is shown in Fig 27. The code block $\mathbb{B}_0$ is certified separately; its precondition merely requires that $\mathbb{B}_0$ itself matches the memory. All the other code including the original `loop` body and every generated one are parameterized as a code block family and certified altogether. In their preconditions, besides the requirement that the code block matches the memory, there should exist an integer i ranged between 0 and 5 (both inclusive), such that the first i instructions have been copied properly, and the three registers \$8, \$9, and \$10 are stored with proper values respectively.

6.4 Code Encryption

Code encryption—or more accurately, runtime code decryption — works similarly as runtime code generation, except that the code generator uses encrypted data located in the same memory region.

A simple encryption and decryption example `encrypt.s` adapted from [27] with its specification is shown in Fig 28. The code block $\mathbb{B}_{\mathtt{pg}}$ between the labels `pg` (inclusive) and `pgend` (exclusive) is the program that is going to be encrypted. In this example, $\mathbb{B}_{\mathtt{pg}}$ simply calculates the sum of 1 and 2 and stores the result 3 into the register \$2, as the precondition of `halt` indicates.

```
       {λ𝕊.True}
main:  la   $8, pg
       la   $9, pgend
       li   $10,0xffffffff
       {a1 * blk(𝔹pg)}
xor1:  lw   $11, 0($8)
       xor  $11, $11, $10
       sw   $11, 0($8)
       addi $8, $8, 4
       blt  $8, $9, xor1
       j    decr

       {λ𝕊.True} 𝔹pg =
pg:    li   $2, 1
       li   $3, 2
       add  $2, $2, $3
       j    halt
pgend:
```

```
       {λ𝕊.True}
decr:  la   $8, pg
       la   $9, pgend
       la   $10,0xffffffff
       {a1 * a2}
xor2:  lw   $11, 0($8)
       xor  $11, $11, $10
       sw   $11, 0($8)
       addi $8, $8, 4
       blt  $8, $9, xor1
       j    pg

       {λ(𝕄,ℝ).ℝ($2) = 3}
halt:  j    halt
```

$\mathtt{a}_1 \triangleq \lambda(\mathbb{M},\mathbb{R}).\mathtt{pg}\le\mathbb{R}(\$8)<\mathtt{pgend}\wedge\mathbb{R}(\$9)=\mathtt{pgend}\wedge\mathbb{R}(\$10)=\mathtt{0xffffffff}$

$\mathtt{a}_2 \triangleq \lambda(\mathbb{M},\mathbb{R}).\left(\mathtt{blk}(\mathbb{B}_{\mathtt{pg}})\ \{\mathtt{pg}\rightsquigarrow\overline{\mathbb{M}(\mathtt{pg})},\ldots,\mathtt{pgend}-1\rightsquigarrow\overline{\mathbb{M}(\mathtt{pgend}-1)}\}\right)$

Figure 28. `encrypt.s`: Code and specification

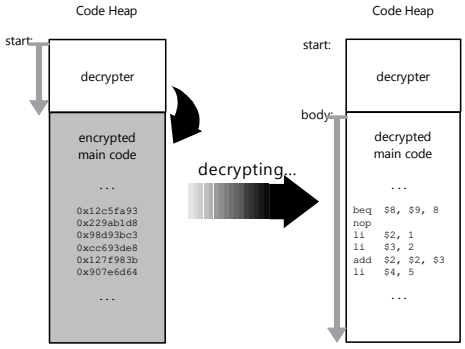

Figure 29. The execution of runtime code decryption

The `main` block together with the `xor1` block are the encryption routine, which flips all the bits stored between `pg` and `pgend`, thus results in a encrypted form. The `decr` block and `xor2` block, on the other hand, will decrypt the encrypted data and obtaining the original code. In addition to the requirement that proper values are stored in the registers \$8 to \$10, `xor1` needs the precondition that $\mathbb{B}_{\mathtt{pg}}$ is properly stored, while `xor2` on the contrary needs to make sure that the flip of $\mathbb{B}_{\mathtt{pg}}$ is properly stored (as $\mathtt{a}_2$ describes).

The encryption and the decryption routines are independent and can be separately executed: one can do the encryption first and store the encrypted code together with the dynamic decryption program, so that at the next time the program is loaded, the code can be decrypted and executed dynamically, as shown in Fig 29.

By making use of parametric code, It is possible to certify this encrypt-decrypter even without knowledge of the content of $\mathbb{B}_{\mathtt{pg}}$. That is, we abstract $\mathbb{B}_{\mathtt{pg}}$ out as a parameter, and prove the general property of the main code : given any code block $\mathbb{B}_{\mathtt{pg}}$, as long as it can be safely executed under certain precondition, the whole combined code is safe to execute under the same precondition; and if $\mathbb{B}_{\mathtt{pg}}$ is properly stored before the encryption, it will still be properly stored after the encryption-decryption cycle. More over, the combined code behaves just the same as $\mathbb{B}_{\mathtt{pg}}$ (meaning that they are both well-formed with respect to the same precondition and specification).

7. Related Work and Conclusion

Previous assembly code certification systems (e.g., TAL [22], FPCC [1, 10], and CAP [31, 24]) all treat code separately from data memory, so that only immutable code is supported. Appel *et al* [2, 21] described a model that treats machine instructions as data in von Neumann style; they raised the verification of runtime code generation as an open problem, but did not provide a solution. TALT [6] also implemented the von Neumann machine model where code is stored in memory, but it still does not support SMC.

TAL/T [13, 29] is a typed assembly language that provides some limited capabilities for manipulating code at runtime. TAL/T code is compiled from Cyclone—a type safe C subset with extra support for template-based runtime code generation. However, since the code generation is implemented by specific macro instructions, it does not support any code modification at runtime.

Otherwise there was actually very little work done on the certification of self-modifying code in the past. Previous program verification systems—including Hoare logic, type system, and proof-carrying code [23]—consistently maintain the assumption that program code stored in memory is immutable.

We have developed a simple Hoare-style framework for modularly verifying general von Neumann machine programs, with strong support for self-modifying code. By statically specifying and reasoning about the possible runtime code sequences, we can now successfully verify arbitrary runtime code modification and/or generation. Taking a unified view of code and data has given us some surprising benefits: we can now apply separation logic to support local reasoning on both program code and regular data structures.

Acknowledgment

We thank Xinyu Feng, Zhaozhong Ni, Hai Fang, and anonymous referees for their suggestions and comments on an earlier version of this paper. Hongxu Cai's research is supported in part by the National Natural Science Foundation of China under Grant No. 60553001 and by the National Basic Research Program of China under Grants No. 2007CB807900 and No. 2007CB807901. Zhong Shao and Alexander Vaynberg's research is based on work supported in part by gifts from Intel and Microsoft, and NSF grant CCR-0524545. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] A. W. Appel. Foundational proof-carrying code. In *Proc. 16th IEEE Symp. on Logic in Computer Science*, pages 247–258, June 2001.

[2] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proc. 27th ACM Symposium on Principles of Programming Languages*, pages 243–253, Jan. 2000.

[3] D. Aucsmith. Tamper resistant software: An implementation. In *Proceedings of the First International Workshop on Information Hiding*, pages 317–333, London, UK, 1996. Springer-Verlag.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proc. 2000 ACM Conf. on Prog. Lang. Design and Implementation*, pages 1–12, 2000.

[5] H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code (extended version & coq implementation). Technical Report YALEU/DCS/TR-1379, Yale Univ., Dept. of Computer Science, Mar. 2007. `http://flint.cs.yale.edu/publications/smc.html`.

[6] K. Crary. Toward a foundational typed assembly language. In *Proc. 30th ACM Symposium on Principles of Programming Languages*, pages 198–212, Jan. 2003.

[7] S. Debray and W. Evans. Profile-guided code compression. In *Proc. 2002 ACM Conf. on Prog. Lang. Design and Implementation*, pages 95–105, New York, NY, 2002.

[8] R. W. Floyd. Assigning meaning to programs. *Communications of the ACM*, Oct. 1967.

[9] N. Glew and G. Morrisett. Type-safe linking and modular assembly language. In *Proc. 26th ACM Symposium on Principles of Programming Languages*, pages 250–261, Jan. 1999.

[10] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Proc. 17th Annual IEEE Symp. on Logic in Computer Science*, pages 89–100, July 2002.

[11] G. M. Henry. Flexible high-performance matrix multiply via a self-modifying runtime code. Technical Report TR-2001-46, Department of Computer Sciences, The University of Texas at Austin, Dec. 2001.

[12] C. A. R. Hoare. Proof of a program: FIND. *Communications of the ACM*, Jan. 1971.

[13] L. Hornof and T. Jim. Certifying compilation and run-time code generation. *Higher Order Symbol. Comput.*, 12(4):337–375, 1999.

[14] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. 28th ACM Symposium on Principles of Programming Languages*, pages 14–26, 2001.

[15] Y. Kanzaki, A. Monden, M. Nakamura, and K. ichi Matsumoto. Exploiting self-modification mechanism for program protection. In *COMPSAC '03*, page 170, 2003.

[16] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[17] J. Larus. SPIM: a MIPS32 simulator. v7.3, 2006.

[18] K. Lawton. BOCHS: IA-32 emulator project. v2.3, 2006.

[19] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *Proc. 1996 ACM Conf. on Prog. Lang. Design and Implementation*, pages 137–148. ACM Press, 1996.

[20] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, 1992.

[21] N. G. Michael and A. W. Appel. Machine instruction syntax and semantics in higher order logic. In *International Conference on Automated Deduction*, pages 7–24, 2000.

[22] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[23] G. Necula. Proof-carrying code. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 106–119, New York, Jan. 1997. ACM Press.

[24] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proc. 33rd ACM Symposium on Principles of Programming Languages*, Jan. 2006.

[25] P. Nordin and W. Banzhaf. Evolving turing-complete programs for a register machine with self-modifying code. In *Proc. of the 6th International Conf. on Genetic Algorithms*, pages 318–327, 1995.

[26] B. C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, Cambridge, MA, 2005.

[27] Ralph. Basics of SMC. `http://web.archive.org/web/20010425070215/awc.rejects.net/files/text/sm%c.txt`, 2000.

[28] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proc. 17th IEEE Symp. on Logic in Computer Science*, 2002.

[29] F. M. Smith. *Certified Run-Time Code Generation*. PhD thesis, Cornell University, Jan. 2002.

[30] The Coq Development Team, INRIA. The Coq proof assistant reference manual. The Coq release v8.0, 2004-2006.

[31] D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. *Science of Computer Programming*, 50(1-3):101–127, Mar. 2004.