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.

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 [21] and other application programs [20, 13, 31]. Dynamic code optimization can improve the performance [4, 11] or minimize the code size [7]. Dynamic code encryption [29] 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 [26]. SMC also arises in applications such as just-in-time compiler, dynamic loading and linking, OS bootloaders, binary translation, and virtual machine monitor.

Unfortunately, SMC is also extremely difficult to reason about: existing formal verification techniques—including Hoare logic [8, 12] and type system [27, 23]—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 [33] 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, 30] 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:

	Examples	System	where
Common Basic Constructs	opcode modification	GCAP2	Sec 5.2
	control flow modification	GCAP2	Sec 6.2
	unbounded code rewriting	GCAP2	Sec 6.5
	runtime code checking	GCAP1	Sec 6.4
	runtime code generation	GCAP1	Sec 6.3
	multilevel RCG	GCAP1	Sec 4.1
	self-mutating code block	GCAP2	Sec 6.9
	mutual modification	GCAP2	Sec 6.7
Typical Applications	self-growing code	GCAP2	Sec 6.6
	polymorphic code	GCAP2	Sec 6.8
	code optimization	GCAP2/1	Sec 6.3
	code compression	GCAP1	Sec 6.10
	code obfuscation	GCAP2	Sec 6.9
	code encryption	GCAP1	Sec 6.10
	OS boot loaders	GCAP1	Sec 6.1
	shellcode	GCAP1	Sec 6.11

Table 1. A summary of GCAP-supported applications

- Our GCAP system is the first formal framework that can successfully certify any form of runtime code manipulation—including all the common basic constructs and important applications given in Table 1. 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[18].
- 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 [30]. 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, 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

In this section, we present the abstract machine model and its operational semantics, both of which are formalized inside a mechanized meta logic (Coq [32]). After that, we use an example to demonstrate the key features of a typical self-modifying program.

2.1 SMC on Stock Hardware

Before proceeding further, a few practical issues for implementing SMC on today's stock hardware need to be clarified. First, most CPU nowadays prefetches a certain number of instructions before executing them from the cache. Therefore, instruction modification has to occur long before it becomes visible. Meantime, for backward compatibility, some processors would detect and handle this itself (at the cost of performance). Another issue is that some RISC machines require the use of branch delay slots. In our system, we assume that all these are hidden from programmers at the assembly level (which is true for the SPIM simulator).

There exist more obstacles against SMC at the OS level. Operating systems usually assign a flag to each memory page to protect data from being executed or, code from being modified. But this can often be get around through special techniques. For example, only stacks are allowed to support both the writing access and the execution at the same time under Microsoft Windows, which provides a way to do SMC.

To simplify the presentation, we will no longer consider these effects because they do not affect our core ideas in the rest of this paper. It is important to first sort out the key techniques for reasoning about SMC, while applying our system to deal with specific hardware and OS restrictions will be left as future work.

2.2 Abstract Machine

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.

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 $\mathtt{f}$ 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 $\mathtt{pc}$ 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 $\mathtt{pc}$ 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 $\mathtt{pc}$ 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, 22] 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. This property can be further refined into the following two fundamental requirements of the Next and Npc func-

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

Figure 1. Definition of target machine GTM

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

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

Figure 2. GTM program execution

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

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

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

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

and

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

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

tions:

$$
\begin{aligned}
&\forall \mathbb{M},\mathbb{M}',\mathbb{M}_0,\mathbb{E},\mathbb{E}',\mathsf{pc},\iota.\ \mathbb{M}\bot\mathbb{M}_0 \wedge \mathsf{Next}_{\mathsf{pc},\iota}(\mathbb{M},\mathbb{E}) = (\mathbb{M}',\mathbb{E}') \rightarrow \\
&\qquad \mathbb{M}'\bot\mathbb{M}_0 \wedge \mathsf{Next}_{\mathsf{pc},\iota}(\mathbb{M}\uplus\mathbb{M}_0,\mathbb{E}) = (\mathbb{M}'\uplus\mathbb{M}_0,\mathbb{E}') \qquad (1)
\end{aligned}
$$

$$
\forall \mathsf{pc},\iota,\mathbb{M},\mathbb{M}',\mathbb{E}.\ \mathsf{Npc}_{\mathsf{pc},\iota}(\mathbb{M},\mathbb{E}) = \mathsf{Npc}_{\mathsf{pc},\iota}(\mathbb{M}\uplus\mathbb{M}',\mathbb{E}) \qquad (2)
$$

where

$$
\begin{aligned}
\mathbb{M}\bot\mathbb{M}' &\triangleq \mathsf{dom}(\mathbb{M})\cap\mathsf{dom}(\mathbb{M}')=\emptyset,\\
\mathbb{M}\uplus\mathbb{M}' &\triangleq \mathbb{M}\cup\mathbb{M}' \text{ if } \mathbb{M}\bot\mathbb{M}'.
\end{aligned}
$$

2.3 Specialization

By specializing every component of the machine $\mathcal{M}$ according to different architectures, we obtain different machines instances.

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*.

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

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

$$
\begin{array}{ll}
\mathbb{R}(\mathtt{r}_{AH}) := \mathbb{R}(\mathtt{r}_{AX})\&(255 \ll 8) & \mathbb{R}(\mathtt{r}_{AL}) := \mathbb{R}(\mathtt{r}_{AX})\&255 \\
\mathbb{R}(\mathtt{r}_{BH}) := \mathbb{R}(\mathtt{r}_{BX})\&(255 \ll 8) & \mathbb{R}(\mathtt{r}_{BL}) := \mathbb{R}(\mathtt{r}_{BX})\&255 \\
\mathbb{R}(\mathtt{r}_{CH}) := \mathbb{R}(\mathtt{r}_{CX})\&(255 \ll 8) & \mathbb{R}(\mathtt{r}_{CL}) := \mathbb{R}(\mathtt{r}_{CX})\&255 \\
\mathbb{R}(\mathtt{r}_{DH}) := \mathbb{R}(\mathtt{r}_{DX})\&(255 \ll 8) & \mathbb{R}(\mathtt{r}_{DL}) := \mathbb{R}(\mathtt{r}_{DX})\&255
\end{array}
$$

$$
\begin{array}{rcl}
\mathbb{R}\{\mathtt{r}_{AH} \rightsquigarrow \mathtt{b}\} & := & \mathbb{R}\{\mathtt{r}_{AX} \rightsquigarrow (\mathbb{R}(\mathtt{r}_{AX})\&255 | \mathtt{b} \ll 8)\} \\
\mathbb{R}\{\mathtt{r}_{AL} \rightsquigarrow \mathtt{b}\} & := & \mathbb{R}\{\mathtt{r}_{AX} \rightsquigarrow (\mathbb{R}(\mathtt{r}_{AX})\&(255 \ll 8) | \mathtt{b})\} \\
\mathbb{R}\{\mathtt{r}_{BH} \rightsquigarrow \mathtt{b}\} & := & \mathbb{R}\{\mathtt{r}_{BX} \rightsquigarrow (\mathbb{R}(\mathtt{r}_{BX})\&255 | \mathtt{b} \ll 8)\} \\
\mathbb{R}\{\mathtt{r}_{BL} \rightsquigarrow \mathtt{b}\} & := & \mathbb{R}\{\mathtt{r}_{BX} \rightsquigarrow (\mathbb{R}(\mathtt{r}_{BX})\&(255 \ll 8) | \mathtt{b})\} \\
\mathbb{R}\{\mathtt{r}_{CH} \rightsquigarrow \mathtt{b}\} & := & \mathbb{R}\{\mathtt{r}_{CX} \rightsquigarrow (\mathbb{R}(\mathtt{r}_{CX})\&255 | \mathtt{b} \ll 8)\} \\
\mathbb{R}\{\mathtt{r}_{CL} \rightsquigarrow \mathtt{b}\} & := & \mathbb{R}\{\mathtt{r}_{CX} \rightsquigarrow (\mathbb{R}(\mathtt{r}_{CX})\&(255 \ll 8) | \mathtt{b})\} \\
\mathbb{R}\{\mathtt{r}_{DH} \rightsquigarrow \mathtt{b}\} & := & \mathbb{R}\{\mathtt{r}_{DX} \rightsquigarrow (\mathbb{R}(\mathtt{r}_{DX})\&255 | \mathtt{b} \ll 8)\} \\
\mathbb{R}\{\mathtt{r}_{DL} \rightsquigarrow \mathtt{b}\} & := & \mathbb{R}\{\mathtt{r}_{DX} \rightsquigarrow (\mathbb{R}(\mathtt{r}_{DX})\&(255 \ll 8) | \mathtt{b})\}
\end{array}
$$

Figure 6. $\mathcal{M}_{\text{x86}}$ 8-bit register use

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

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

and

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

Figure 7. $\mathcal{M}_{\text{x86}}$ operational semantics

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. $\mathsf{beq}\ \mathtt{r}_s,\mathtt{r}_t,i$), 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. It is limited to 16-bit real mode, and has only a small subset of instructions, including indirect and conditional jumps. We must note, however, that this is not due

Call 0x13 (disk operations) $\quad (id = 0x13)$

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

Parameters

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

Conditions

$$
\begin{array}{ll}
\text{Cylinder} = 0 & \text{Head} = 0 \\
\text{Disk Id} = 0\text{x}80 & \text{Sector} < 63
\end{array}
$$

Effect

$$
\begin{array}{ll}
\mathbb{M}' = \mathbb{M}\{Dest + i \rightsquigarrow \mathbb{D}(Src + i)\} & (0 \le i \le \text{Bytes}) \\
\mathbb{R}' = \mathbb{R}\{\mathtt{r}_{AH} \rightsquigarrow 0\} & \mathbb{D}' = \mathbb{D}
\end{array}
$$

Figure 8. 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 9. `opcode.s`: Opcode modification

to inability to add such instructions, but simply due to the lack of time that would be needed to be both extensive and correct in our definitions. But the subset is not so trivial as to be useless; in fact it is adequate for certification of interesting examples such as OS boot loaders.

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. Some of the features of $\mathcal{M}_{\text{x86}}$ include the 8-bit registers, which are dependent on 16-bit registers. This fact that is responsible for Fig 6, which shows how the 8-bit registers are extracted out of the 16-bit ones. The memory of $\mathcal{M}_{\text{x86}}$ is segmented, a fact which is mostly invisible, except in the long jump instruction (Fig 7). as a black box with proper formal specifications (Fig 8). 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 the rules of the BIOS semantics are large, it is unwieldy to present them in the usual mathematical notation, and instead a more descriptive form is used. The precise definition can be extracted if one takes the parameters to be let statements, the condition and the command number to be a conjunctive predicate over the initial state, and the effect to be the ending state in the form $(\mathbb{M}', \mathbb{R}', \mathbb{D}')$.

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. The BIOS disk read command uses that assumption.

2.4 A Taste of SMC

We give a sample piece of self-modifying code (i.e., `opcode.s`) in Fig 9. 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 $\mathsf{addi}\ \$2, \$2, 1$. 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) [32] as our meta logic. Our implementation is done using Coq [32] 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:

$$
\begin{array}{rl}
(Term)\ A, B ::= & \mathsf{Set} \mid \mathsf{Prop} \mid \mathsf{Type} \mid x \mid \lambda x{:}A.\, B \mid A\ B \\
& \mid A \to B \mid \textit{ind. def.} \mid \ldots \\
(Prop)\ p, q ::= & \mathsf{True} \mid \mathsf{False} \mid \neg p \mid p \wedge q \mid p \vee q \mid p \to q \\
& \mid \forall x{:}A.\, p \mid \exists x{:}A.\, p \mid \ldots
\end{array}
$$

3.1 From Invariants to Hoare Logic

The program safety predicate can be defined as follows:

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

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

Safen_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 [17] 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 $\forall \mathbb{W}.\, \mathsf{Inv}(\mathbb{W}) \to \mathsf{Safe}(\mathbb{W})$.

Invariant-based method can be directly used to prove the example we just presented in Fig 9: one can construct a global invariant for the code, as much like the tedious one in Fig 10, which satisfies our Definition 3.1.

Invariant-based verification is powerful especially when carried out with a rich meta logic. But a global invariant for a program, usually troublesome and complicated, is often unrealistic to directly find and write down. A global invariant needs to specify the condition on every memory address in the code. Even worse, this prevents separate verification (of different components) and local reasoning, as we see: every program module has to be annotated with the same global invariant which requires the knowledge of the entire code base.

$$
\begin{aligned}
&\mathsf{Inv}(\mathbb{M},\mathbb{R},\mathsf{pc}) \triangleq \mathtt{Decode}(\mathbb{M},100,\iota_{100}) \wedge \mathtt{Decode}(\mathbb{M},200,\iota_{200}) \wedge \\
&\quad \mathtt{Decode}(\mathbb{M},208,\iota_{208}) \wedge \mathtt{Decode}(\mathbb{M},212,\iota_{212}) \wedge \mathtt{Decode}(\mathbb{M},216,\iota_{216}) \wedge \\
&\quad \mathtt{Decode}(\mathbb{M},224,\iota_{224}) \wedge \mathtt{Decode}(\mathbb{M},232,\iota_{232}) \wedge (\\
&\quad (\mathsf{pc}{=}200 \wedge \mathtt{Decode}(\mathbb{M},204,\iota_{204})) \vee (\mathsf{pc}{=}204 \wedge \mathtt{Decode}(\mathbb{M},204,\iota_{204})) \vee \\
&\quad (\mathsf{pc}{=}204 \wedge \mathtt{Decode}(\mathbb{M},204,\iota_{100}) \wedge \mathbb{R}(\$2) = \mathbb{R}(\$4)) \vee \\
&\quad (\mathsf{pc}{=}208 \wedge \mathbb{R}(\$4) \le \mathbb{R}(\$2) \le \mathbb{R}(\$4)+1) \vee \\
&\quad (\mathsf{pc}{=}212 \wedge \mathbb{R}(\$4) \le \mathbb{R}(\$2) \le \mathbb{R}(\$4)+1) \vee (\mathsf{pc}{=}216 \wedge \mathbb{R}(\$2) \ne \mathbb{R}(\$4)) \vee \\
&\quad (\mathsf{pc}{=}224 \wedge \mathbb{R}(\$2) = \mathbb{R}(\$4) \wedge \langle \mathbb{R}(\$9) \rangle_4 = \mathsf{Ec}(\iota_{100})) \vee \\
&\quad (\mathsf{pc}{=}232 \wedge \mathbb{R}(\$2) = \mathbb{R}(\$4) \wedge \mathtt{Decode}(\mathbb{M},204,\iota_{100})))
\end{aligned}
$$

Figure 10. A global invariant for the opcode example

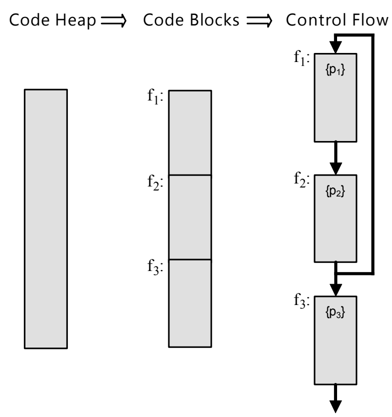

Figure 11. Hoare-style reasoning of assembly code

Traditional Hoare-style reasoning over assembly programs (e.g., CAP [33]) is illustrated in Fig 11. 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. Comparing to the limited usage of existing systems TAL or CAP, a system working on GTM do have several advantages:

First, instruction set and operational semantics come as an integrated part for a TAL or CAP system, thus are usually fixed and limited. Although extensions can be made, it costs fair amount of efforts. On the other hand, GTM abstracts both components out, and a system that directly works on it will be robust and suitable for different architecture.

Also, GTM is more realistic since it has the benefit that programs are encoded and stored in memory as real computing platform does. Besides regular assembly code, A GTM-based verification system would be able to manage real binary code.

Our generalization 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

$$
\begin{array}{rrl}
(\mathit{CodeBlock}) & \mathbb{B} & ::= \mathtt{f}:\mathbb{I} \\
(\mathit{InstrSeq}) & \mathbb{I} & ::= \iota;\mathbb{I} \mid \iota \\
(\mathit{CodeHeap}) & \mathbb{C} & ::= \{\mathtt{f} \leadsto \mathbb{I}\}^{*} \\
(\mathit{Assertion}) & \mathsf{a} & \in \mathit{State} \to \mathit{Prop} \\
(\mathit{ProgSpec}) & \Psi & \in \mathit{Address} \rightharpoonup \mathit{Assertion}
\end{array}
$$

Figure 12. Auxiliary constructs and specifications

$$
\begin{array}{llll}
\mathsf{a} \Rightarrow \mathsf{a}' & \triangleq \forall \mathbb{S}.(\mathsf{a}\ \mathbb{S} \to \mathsf{a}'\mathbb{S}) & \mathsf{a} \Leftrightarrow \mathsf{a}' & \triangleq \forall \mathbb{S}.(\mathsf{a}\ \mathbb{S} \leftrightarrow \mathsf{a}'\mathbb{S}) \\
\neg \mathsf{a} & \triangleq \lambda \mathbb{S}.\neg \mathsf{a}\ \mathbb{S} & \mathsf{a} \wedge \mathsf{a}' & \triangleq \lambda \mathbb{S}.\mathsf{a}\ \mathbb{S} \wedge \mathsf{a}'\mathbb{S} \\
\mathsf{a} \vee \mathsf{a}' & \triangleq \lambda \mathbb{S}.\mathsf{a}\ \mathbb{S} \vee \mathsf{a}'\mathbb{S} & \mathsf{a} \to \mathsf{a}' & \triangleq \lambda \mathbb{S}.\mathsf{a}\ \mathbb{S} \to \mathsf{a}'\mathbb{S} \\
\forall x.\mathsf{a} & \triangleq \lambda \mathbb{S}.\forall x.(\mathsf{a}\ \mathbb{S}) & \exists x.\mathsf{a} & \triangleq \lambda \mathbb{S}.\exists x.(\mathsf{a}\ \mathbb{S}) \\
\mathsf{a} * \mathsf{a}' & \multicolumn{3}{l}{\triangleq \lambda(\mathbb{M}_0,\mathbb{R}).\exists \mathbb{M},\mathbb{M}'.\mathbb{M}_0{=}\mathbb{M}\uplus\mathbb{M}' \wedge \mathsf{a}(\mathbb{M},\mathbb{R}) \wedge \mathsf{a}'(\mathbb{M}',\mathbb{R})}
\end{array}
$$

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

Figure 13. Assertion operators

immutability can be enforced in the inference rules using a simple separation conjunction borrowed from separation logic [14, 30].

3.2 Specification Language

Our specification language is defined in Fig 12. A code block $\mathbb{B}$ is a syntactic unit that represents a sequence $\mathbb{I}$ of instructions, beginning at specific memory address $\mathtt{f}$. 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 13 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 $\mathsf{a} \to \mathsf{a}'$ and $\mathsf{a} \Rightarrow \mathsf{a}'$: the former is an assertion, while the latter is a proposition! We also define standard separation logic primitives [14, 30] 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. Solid work has been established on separation logic, which we use heavily in our proofs.

Lemma 3.3 (Properties for separation logic) Let a, a' and a'' be assertions, then we have

1. For any assertions a and a', we have $\mathsf{a} * \mathsf{a}' \Leftrightarrow \mathsf{a}' * \mathsf{a}$.
2. For any assertions a, a' and a'', we have $(\mathsf{a} * \mathsf{a}') * \mathsf{a}'' \Leftrightarrow \mathsf{a} * (\mathsf{a}' * \mathsf{a}'')$.
3. Given type T, for any assertion function $P \in T \to \mathit{Assertion}$ and assertion a, we have $(\exists x \in T.P\ x) * \mathsf{a} \Leftrightarrow \exists x \in T.(P\ x * \mathsf{a})$.
4. Given type T, for any assertion function $P \in T \to \mathit{Assertion}$ and assertion a, we have $(\forall x \in T.P\ x) * \mathsf{a} \Leftrightarrow \forall x \in T.(P\ x * \mathsf{a})$.

We omit the proof of these properties since they are standard laws of separation logic [14, 30].

$$
\begin{array}{lll}
\mathtt{blk}(\mathtt{f}:\mathbb{I}) & \triangleq & \begin{cases} \lambda\mathbb{S}.\mathtt{Decode}(\mathbb{S},\mathtt{f},\iota) & \text{if } \mathbb{I}=\iota \\ \lambda\mathbb{S}.\mathtt{Decode}(\mathbb{S},\mathtt{f},\iota)\wedge(\mathtt{blk}(\mathtt{f}{+}|\mathsf{Ec}(\iota)|:\mathbb{I}')\ \mathbb{S}) & \text{if } \mathbb{I}=\iota;\mathbb{I}' \end{cases} \\
\mathtt{blk}(\mathbb{C}) & \triangleq & \forall \mathtt{f}\in\mathsf{dom}(\mathbb{C}).\,\mathtt{blk}(\mathtt{f}:\mathbb{C}(\mathtt{f})) \\
\Psi\Rightarrow\Psi' & \triangleq & \forall \mathtt{f}\in\mathsf{dom}(\Psi).\,(\Psi(\mathtt{f})\Rightarrow\Psi'(\mathtt{f})) \\
(\Psi_1\cup\Psi_2)(\mathtt{f}) & \triangleq & \begin{cases} \Psi_1(\mathtt{f}) & \text{if } \mathtt{f}\in\mathsf{dom}(\Psi_1)\setminus\mathsf{dom}(\Psi_2) \\ \Psi_2(\mathtt{f}) & \text{if } \mathtt{f}\in\mathsf{dom}(\Psi_2)\setminus\mathsf{dom}(\Psi_1) \\ \Psi_1(\mathtt{f})\vee\Psi_2(f) & \text{if } \mathtt{f}\in\mathsf{dom}(\Psi_1)\cap\mathsf{dom}(\Psi_2) \end{cases}
\end{array}
$$

Figure 14. Predefined functions

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

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

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

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

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

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

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

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

Figure 15. Inference rules for GCAP0

Fig 14 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. We also define two operators between program specifications. We say that one program specification is stricter than another, namely $\Psi\Rightarrow\Psi'$, if every assertion in Ψ implies the corresponding assertion at the same address in Ψ'. The union of two program specifications is just the disjunction of the two corresponding assertions at each address. Clearly, any two program specifications are stricter than their union specification.

3.3 Inference Rules

Fig 15 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 ($\Psi\vdash\{\mathsf{a}\}\mathbb{B}$) iff, starting from a state satisfying its precondition 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 -) directly follows this understanding. Inductively, to validate the well-formedness of a code block beginning with ι under precondition a (rule), we should find an intermediate assertion a' serving simultaneously as the precondition of the tail code sequence, and the postcondition of ι. In the second premise of , since our syntax does not have a postcondition, 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 and rules on each instruction if necessary. For example, specializing over the direct jump ($\mathsf{j}\ \mathtt{f}'$) results in the following rule:

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

Specializing over the add instruction makes

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

which via can be further reduced into

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

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

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

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 rule casts a code block into a corresponding well-formed singleton code heap, and the - 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 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 a is satisfied, separately from the code section;
- the instruction sequence is well-formed under a.

The 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 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 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. This is established through a series of weakening lemmas, then progress and preservation lemmas; frame rules are also easy to prove.

Lemma 3.4 $\Psi\vdash\mathbb{C}:\Psi'$ if and only if for every $\mathtt{f}\in\mathsf{dom}(\Psi')$ we have $\mathtt{f}\in\mathsf{dom}(\mathbb{C})$ and $\Psi\vdash\{\Psi'(\mathtt{f})\}\mathtt{f}:\mathbb{C}(\mathtt{f})$.

Proof Sketch: For the necessity, prove from inversion of the -

rule, the rule, and the - rule. For the sufficiency, just repetitively apply the - rule. ■

Lemma 3.5 contains the standard weakening rules and can be proved by induction over the well-formed-code-block rules and by using Lemma 3.4.

Lemma 3.5 (Weakening Properties)

$$\frac{\Psi \vdash \{a\}\mathbb{B} \quad \Psi \Rightarrow \Psi' \quad a' \Rightarrow a}{\Psi' \vdash \{a'\}\mathbb{B}} \quad (\quad - \quad)$$

$$\frac{\Psi_1 \vdash \mathbb{C} : \Psi_2 \quad \Psi_1 \Rightarrow \Psi_1' \quad \Psi_2' \Rightarrow \Psi_2}{\Psi_1' \vdash \mathbb{C} : \Psi_2'} \quad (\quad - \quad)$$

Proof:

1. By doing induction over the rule and the rule, we have the result.
2. Easy to see from 1 and Lemma 3.4. ■

Lemma 3.6 (Progress) If $\Psi \vdash \mathbb{W}$, then there exists a program $\mathbb{W}'$, such that $\mathbb{W} \longmapsto \mathbb{W}'$.

Proof: By inversion of the rule we know that there is a code sequence $\mathbb{I}$ indeed in the memory starting from the current $\mathtt{pc}$. Suppose the first instruction of $\mathbb{I}$ is ι, then from the rules for well-formed code blocks, we learn that there exists Ψ', such that $\Psi' \vdash \{a\}\iota$.

Since a is satisfied for the current state, by inversion of the rule, this guarantees that our world $\mathbb{W}$ is safe to be executed further for at least one step. ■

Lemma 3.7 (Preservation) If $\Psi \vdash \mathbb{W}$ and $\mathbb{W} \longmapsto \mathbb{W}'$, then $\Psi \vdash \mathbb{W}'$.

Proof Sketch: Suppose the premises of the rule hold. Again, a code block $\mathtt{pc} : \mathbb{I}$ is present in the current memory. We only need to find an a' and $\mathbb{I}'$ satisfying $(a' * (\mathtt{blk}(\mathbb{C}) \wedge \mathtt{blk}(\mathtt{pc} : \mathbb{I}')) \ \mathbb{S}$ and $\Psi \vdash \{a'\}\mathtt{pc} : \mathbb{I}'$, when there is a $\mathbb{W}' = (\mathbb{M}', \mathbb{E}', \mathtt{pc}')$ such that $\mathbb{W} \longmapsto \mathbb{W}'$.

There are two cases:

1. $\mathbb{I} = \iota$ is a single-instruction sequence. It would be easy to show that $\mathtt{pc}' \in \mathsf{dom}(\Psi)$ and $\mathtt{pc}' \in \mathsf{dom}(\mathbb{C})$. We choose
$$a' = \Psi(\mathtt{pc}'), \text{ and } \mathbb{I}' = \mathbb{C}(\mathtt{pc}')$$
to satisfy the two conditions.
2. $\mathbb{I} = \iota;\mathbb{I}'$ is a multi-instruction sequence. Then by inversion of the rule, either we reduce to the previous case, or $\mathtt{pc}' = \mathtt{pc} + |\mathsf{Ec}(\iota)|$ and there is an a' such that $\Psi \vdash \{a'\}\mathtt{pc}' : \mathbb{I}'$. Thus a' and $\mathbb{I}'$ are what we choose to satisfy the two conditions. ■

Theorem 3.8 (Soundness of GCAP0) If $\Psi \vdash \mathbb{W}$, then $\mathsf{Safe}(\mathbb{W})$.

Proof: Define predicate $\mathsf{Inv} \triangleq \lambda\mathbb{W}.(\Psi \vdash \mathbb{W})$. Then by Lemma 3.6 and Lemma 3.7, Inv is an invariant of GTM. Together with Theorem 3.2, the conclusion holds. ■

The following lemma (a.k.a., the frame rule) captures the essence of local reasoning for separation logic:

Lemma 3.9
$$\frac{\Psi \vdash \{a\}\mathbb{B}}{(\lambda f.\Psi(f) * a') \vdash \{a * a'\}\mathbb{B}} \quad (\quad - \quad)$$
where a' is independent of every register modified by $\mathbb{B}$.

$$\frac{\Psi \vdash \mathbb{C} : \Psi'}{(\lambda f.\Psi(f) * a) \vdash \mathbb{C} : \lambda f.\Psi'(f) * a} \quad (\quad - \quad)$$

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

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

$$\frac{\begin{array}{c}\Psi \vdash (\mathbb{C}, \Psi) \qquad \mathbb{C}' \subseteq \mathbb{C} \\ (a * (\mathtt{blk}(\mathbb{C}') \wedge \mathtt{blk}(\mathtt{pc} : \mathbb{I}))) \ \mathbb{S} \qquad \Psi' \vdash \{a\}\mathtt{pc} : \mathbb{I} \\ \forall f \in \mathsf{dom}(\Psi').(\Psi'(f) * (\mathtt{blk}(\mathbb{C}') \wedge \mathtt{blk}(\mathtt{pc} : \mathbb{I})) \Rightarrow \Psi(f))\end{array}}{\Psi \vdash (\mathbb{S}, \mathtt{pc})} \quad (\quad - \quad)$$

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

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

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

Figure 16. Inference rules for GCAP1

Proof: For the first rule, we need the following important fact of our GTM specializations:

If $\mathbb{M}_0 \bot \mathbb{M}_1$ and

$$\mathsf{Next}_{\mathtt{pc},\iota}(\mathbb{M}_0, \mathbb{E}) = (\mathbb{M}_0', \mathbb{E}'),$$

then we have $\mathbb{M}_0' \bot \mathbb{M}_1$, and

$$\mathsf{Next}_{\mathtt{pc},\iota}(\mathbb{M}_0 \uplus \mathbb{M}_1, \mathbb{E}) = (\mathbb{M}_0' \uplus \mathbb{M}_1, \mathbb{E}')$$

Then it is easy to prove by induction over the two well-formed code block rules. The second rule is thus straightforward. ■

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 [30].

With the - 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 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 15). To support runtime code generation, we change the top rule and insert an extra layer of judgments called well-formed code specification (see Fig 16) 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 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. - rule has the same form as - , except that it works on the dynamic layer.

The new top rule (-) 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.

Lemma 4.1 $\dfrac{\Psi \vdash (\mathbb{C}, \Psi')}{(\lambda \mathtt{f}.\Psi(\mathtt{f}) * \mathtt{a}) \vdash (\mathbb{C}, \lambda \mathtt{f}.\Psi'(\mathtt{f}) * \mathtt{a})}$ (-)
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 - rule, the conclusion follows immediately from the induction hypothesis; if the final step is via the rule, it must be derived from a well-formed-code-heap derivation:

$$\Psi_0 \vdash \mathbb{C} : \Psi_0' \tag{3}$$

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 - rule to (3) 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 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 - 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 - 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 3.8 (see TR [5] for more detail).

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

This theorem can be established following a similar technique as in GCAP0 (more cases need to be analyzed because of the additional layer). However, we decide to delay the proof to the next section to give better understanding of the relationship between GCAP1 and our more general framework GCAP2. By converting a well-formed GCAP1 program to a well-formed GCAP2 program, we will finally see that GCAP1 is sound given the fact that GCAP2 is sound (see Theorem 5.5 and Theorem 5.8 in the next section).

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 - rule to establish the initial state and prove the correctness of the entire program.

The original code:

```
     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)
B1          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 17. `mrcg.s`: Multilevel runtime code generation

4.1 Example: Multilevel Runtime Code Generation

We use a small example `mrcg.s` in Fig 17 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:

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

where

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

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

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

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 rule and the 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 $\mathtt{blk}(\mathbb{B}_1)$ 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.

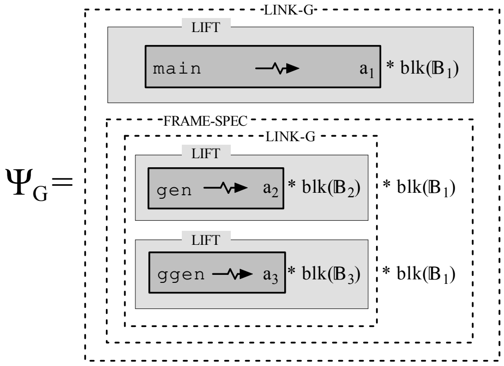

Figure 18. `mrcg.s`: GCAP1 specification

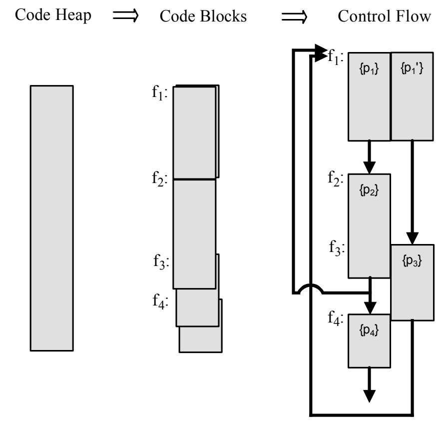

Figure 19. Typical Control Flow in GCAP2

After the three code blocks are locally certified, the rule and then the rule are respectively applied to each of them, as illustrated in Fig 18, resulting in three well-formed singleton code heaps. Afterwards, $\mathbb{B}_2$ and $\mathbb{B}_3$ are linked together and we apply - 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 18 indicates):

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

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 - 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 9 at the end of Sec 2. 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.

5.1 Main Development

The main idea of GCAP2 is illustrated in Fig 19. 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 $\texttt{f}_1$ in Fig 19) 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. $\texttt{f}_1$ in Fig 19), there should *exist* at least one code block there, whose precondition is matched. After all the code blocks are certified, 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. Sec 6.9 discusses more about the importance of this generalization.

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

Figure 20. Assertion language for GCAP2

Specification language. The specification language is almost same as GCAP1, but GCAP2 introduces one new concept called *code specification* (denoted as Φ in Fig 20), 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 21): 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 ( ) has one more requirement that the instruction must actually be in the proper location of memory. Previously in GCAP1, this is guaranteed by the 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 $\Psi \vdash \{\texttt{a}\}\texttt{f} : \mathbb{I}$ now states that if a machine state satisfies assertion $\texttt{a}$, then $\mathbb{I}$ is the only possible code sequence to be executed starting from $\texttt{f}$, until we get to a program point where the specification Ψ can be matched.

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

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

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

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

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

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

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

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

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

Figure 21. Inference rules for GCAP2

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. An example is as follows:

```
{blk(100: movi r0,Ec(j 100); sw r0,102)}
100:  movi r0, Ec(j 100)
      sw   r0, 102
      j    100
```

The third instruction is actually generated by the first two, thus does not appear in the precondition. This kind of judgment can never be shown in GCAP1, nor in any previous Hoare-logic models.

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 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), 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. The soundness proof follows almost the same techniques as in GCAP0.

Firstly, due to the similarity between the rules for well-formed code blocks, the same weakening lemmas still hold:

Lemma 5.1

$$\frac{\Psi \vdash \{\mathtt{a}\}\mathbb{B} \quad \Psi \Rightarrow \Psi' \quad \mathtt{a}' \Rightarrow \mathtt{a}}{\Psi' \vdash \{\mathtt{a}'\}\mathbb{B}} \ (\qquad - \qquad)$$

$$\frac{\Psi \vdash \Phi \quad \Psi \Rightarrow \Psi' \quad \forall \mathbb{B} \in \mathsf{dom}(\Phi'). (\Phi'\ \mathbb{B} \Rightarrow \Phi\ \mathbb{B})}{\Psi' \vdash \Phi'} \ (\qquad - \quad)$$

And also we have the relation between well-formed code specification and well-formed code blocks.

Lemma 5.2 $\Psi \vdash \Phi$ if and only if for every $\mathbb{B} \in \mathsf{dom}(\Phi)$ we have $\Psi \vdash \{\Phi\ \mathbb{B}\}\mathbb{B}$.

Proof Sketch: For the necessity, prove from inversion of and rule, and weaken properties (Lemma 5.1). The sufficiency is trivial by rule. ■

Lemma 5.3 (Progress) If $\Psi \vdash \mathbb{W}$, then there exists a program $\mathbb{W}'$, such that $\mathbb{W} \longmapsto \mathbb{W}'$.

Proof Sketch: Easy to see by inversion of the rule, the rule and the rule successively. ■

Lemma 5.4 (Preservation) If $\Psi \vdash \mathbb{W}$ and $\mathbb{W} \longmapsto \mathbb{W}'$, then $\Psi \vdash \mathbb{W}'$.

Proof Sketch: The first premise is already guaranteed. The other two premises can be verified by checking the two cases of well-formed code block rules. ■

Theorem 5.5 (Soundness of GCAP2) If $\Psi \vdash \mathbb{W}$, then $\mathsf{Safe}(\mathbb{W})$.

Local reasoning. Frame rules are still the key idea for supporting local reasoning.

Theorem 5.6 (Frame Rules)

$$\frac{\Psi \vdash \{\mathtt{a}\}\mathbb{B}}{(\lambda \mathtt{f}. \Psi(\mathtt{f}) * \mathtt{a}') \vdash \{\mathtt{a} * \mathtt{a}'\}\mathbb{B}} \ (\qquad - \qquad)$$

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

$$\frac{\Psi \vdash \Phi}{(\lambda \mathtt{f}. \Psi(\mathtt{f}) * \mathtt{a}) \vdash (\lambda \mathbb{B}. \Phi\ \mathbb{B} * \mathtt{a})} \ (\qquad - \quad)$$

where $\mathtt{a}$ is independent of every register modified by any code block in the domain of Φ.

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 rule.

5.2 Example and Discussion

We can now use GCAP2 to certify the opcode-modification example given in Fig 9. There are four runtime code blocks that need to be handled. Fig 22 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 22 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

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

Let

$$\mathsf{a}_1 \triangleq \mathtt{blk}(\mathtt{new}: \mathsf{addi}\ \$2,\$2,1) * \mathtt{blk}(\mathbb{B}_1)$$
$$\mathsf{a}'_1 \triangleq \mathtt{blk}(\mathbb{B}_3) * \mathtt{blk}(\mathbb{B}_4)$$
$$\mathsf{a}_2 \triangleq (\lambda(\mathbb{M},\mathbb{R}).\,\mathbb{R}(\$2){=}\mathbb{R}(\$4)) * \mathtt{blk}(\mathbb{B}_2)$$
$$\mathsf{a}_3 \triangleq (\lambda(\mathbb{M},\mathbb{R}).\,\mathbb{R}(\$4) \le \mathbb{R}(\$2) \le \mathbb{R}(\$4)+1)$$
$$\mathsf{a}_4 \triangleq (\lambda(\mathbb{M},\mathbb{R}).\,\mathbb{R}(\$2){=}\mathbb{R}(\$4)) * \mathtt{blk}(\mathtt{new}: \mathsf{addi}\ \$2,\$2,1) * \mathtt{blk}(\mathbb{B}_1)$$

Then local judgments are as follows

$$\{\mathtt{modify} \rightsquigarrow \mathsf{a}_4, \mathtt{halt} \rightsquigarrow \mathsf{a}_3\} \vdash \{\mathsf{a}_1\}\mathbb{B}_1$$
$$\{\mathtt{halt} \rightsquigarrow \mathsf{a}_3\} \vdash \{\mathsf{a}_2\}\mathbb{B}_2$$
$$\{\mathtt{halt} \rightsquigarrow \mathsf{a}_3 * \mathtt{blk}(\mathbb{B}_3)\} \vdash \{\mathsf{a}_3 * \mathtt{blk}(\mathbb{B}_3)\}\mathbb{B}_3$$
$$\{\mathtt{target} \rightsquigarrow \mathsf{a}_2\} \vdash \{\mathsf{a}_4 * \mathtt{blk}(\mathbb{B}_4)\}\mathbb{B}_4$$

After applying frame rules and linking, the global specification becomes

$$\{\mathtt{modify} \rightsquigarrow \mathsf{a}_4 * \mathsf{a}'_1, \mathtt{halt} \rightsquigarrow \mathsf{a}_3 * \mathsf{a}'_1\} \vdash \{\mathsf{a}_1 * \mathsf{a}'_1\}\mathbb{B}_1$$
$$\{\mathtt{halt} \rightsquigarrow \mathsf{a}_3 * \mathtt{blk}(\mathbb{B}_3)\} \vdash \{\mathsf{a}_2 * \mathtt{blk}(\mathbb{B}_3)\}\mathbb{B}_2$$
$$\{\mathtt{halt} \rightsquigarrow \mathsf{a}_3 * \mathtt{blk}(\mathbb{B}_3)\} \vdash \{\mathsf{a}_3 * \mathtt{blk}(\mathbb{B}_3)\}\mathbb{B}_3$$
$$\{\mathtt{target} \rightsquigarrow \mathsf{a}_2 * \mathtt{blk}(\mathbb{B}_3)\} \vdash \{\mathsf{a}_4 * \mathsf{a}'_1\}\mathbb{B}_4$$

Figure 22. `opcode.s`: Code and specification

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 rule (the union of two program specifications used in the rule is defined in Fig 14).

Finally, all these judgments are joined together via the 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 process is no longer required here. The global code specification is exactly:

$$\Phi_G = \{\mathbb{B}_1 \rightsquigarrow \mathsf{a}_1 * \mathsf{a}'_1, \mathbb{B}_2 \rightsquigarrow \mathsf{a}_2 * \mathtt{blk}(\mathbb{B}_3), \mathbb{B}_3 \rightsquigarrow \mathsf{a}_3 * \mathtt{blk}(\mathbb{B}_3), \mathbb{B}_4 \rightsquigarrow \mathsf{a}_4 * \mathsf{a}'_1\}$$

which satisfies $[\![\Phi_G]\!] \vdash \Phi_G$ and ultimately the 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 . In the example above we need to prove the judgment

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

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 5.7 (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

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

Proof: We give a way to construct Ψ from Inv:

$$\Phi(\mathtt{f}:\iota) = \lambda\mathbb{S}.\,\mathsf{Inv}(\mathbb{S},\mathtt{f}) \wedge \mathtt{Decode}(\mathbb{S},\mathtt{f},\iota), \quad \forall \mathtt{f},\iota$$
$$\Psi = [\![\Phi]\!] = \lambda\mathtt{f}.\,\lambda\mathbb{S}.\,\mathsf{Inv}(\mathbb{S},\mathtt{f})$$

We first prove the forward direction. Given a program $(\mathbb{S},\mathtt{pc})$ satisfying $\mathsf{Inv}(\mathbb{S},\mathtt{pc})$, we want to show that there exists a such that $\Psi \vdash (\mathbb{S},\mathtt{pc})$.

Observe the fact that for all $\mathtt{f}$ and $\mathbb{S}$,

$$\begin{aligned}\Psi(\mathtt{f})\ \mathbb{S} &\longleftrightarrow \exists\mathbb{I}.\,\Phi(\mathtt{f}:\mathbb{I})\ \mathbb{S}\\ &\longleftrightarrow \exists\iota.\,\mathsf{Inv}(\mathbb{S},\mathtt{f}) \wedge \mathtt{Decode}(\mathbb{S},\mathtt{f},\iota)\\ &\longleftrightarrow \mathsf{Inv}(\mathbb{S},\mathtt{f}) \wedge \exists\iota.\,\mathtt{Decode}(\mathbb{S},\mathtt{f},\iota)\\ &\longleftrightarrow \mathsf{Inv}(\mathbb{S},\mathtt{f})\end{aligned}$$

where the last equivalence is due to the Progress property of Inv together with the transition relation of GTM.

Now we prove the first premise of , i.e. $[\![\Phi]\!] \vdash \Phi$, which by rule , is equivalent to

$$\forall \mathbb{B} \in \mathsf{dom}(\Phi).\,([\![\Phi]\!] \vdash \{\Phi\ \mathbb{B}\}\mathbb{B})$$

which is

$$\forall \mathtt{f},\iota.\,(\Psi \vdash \{\Phi(\mathtt{f}:\iota)\}\mathtt{f}:\iota) \tag{4}$$

which can be rewritten as

$$\forall \mathtt{f},\iota.\,(\lambda\mathtt{f}.\,\lambda\mathbb{S}'.\,\mathsf{Inv}(\mathbb{S}',\mathtt{f}) \vdash \{\lambda\mathbb{S}'.\,\mathsf{Inv}(\mathbb{S}',\mathtt{f}) \wedge \mathtt{Decode}(\mathbb{S}',\mathtt{f},\iota)\}\mathtt{f}:\iota) \tag{5}$$

On the other hand, since Inv is an invariant, it should satisfy the progress and preservation properties. Applying rule, (5) can be easily proved. Therefore, $[\![\Phi]\!] \vdash \Phi$.

Now, by the progress property of Inv, there should exist a valid instruction ι such that $\mathtt{Decode}(\mathbb{S},\mathtt{f},\iota)$ holds. Choose

$$\mathsf{a} \triangleq \lambda\mathbb{S}'.\,\mathsf{Inv}(\mathbb{S}',\mathtt{pc}) \wedge \mathtt{Decode}(\mathbb{S}',\mathtt{f},\iota)$$

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

$\mathbb{B}_1$

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

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

$\mathbb{B}_2$

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

Figure 23. `bootloader.s`: Code and specification

Then it's obvious that $\mathtt{a}\ \mathbb{S}$ is true. And since the correctness of $[\![\Phi]\!] \vdash \{\mathtt{a}\}\mathtt{pc} : \iota$ is included in (4), all the premises in rule has been satisfied.

So we get $\Psi \vdash (\mathbb{S}, \mathtt{pc})$, and it's trivial to see that $\Psi(\mathtt{pc})\ \mathbb{S}$.

On the other hand, for our Ψ, if $\Psi(\mathtt{pc})\ \mathbb{S}$, obvious Inv holds on the world $(\mathbb{S}, \mathtt{pc})$. This completes our proof. ■

Together with the soundness theorem (Theorem 5.5), 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. The translation relation is described by the following theorems. Here we use $\vdash_{\text{GCAP1}}$ and $\vdash_{\text{GCAP2}}$ to represent the two kinds of judgements respectively. Let

$$[\![(\{\mathtt{f}_i \rightsquigarrow \mathbb{I}_i\}^*, \{\mathtt{f}_i \rightsquigarrow \mathtt{a}_i\}^*)]\!] \triangleq \{(\mathtt{f}_i : \mathbb{I}_i) \rightsquigarrow \mathtt{a}_1\}^*$$

then we have

Theorem 5.8 (GCAP1 to GCAP2)

1. If $\Psi \vdash_{\text{GCAP0}} \{\mathtt{a}\}\mathbb{B}$, then $(\lambda \mathtt{f}. \Psi(\mathtt{f}) * \mathtt{blk}(\mathbb{B})) \vdash_{\text{GCAP2}} \{\mathtt{a} * \mathtt{blk}(\mathbb{B})\}\mathbb{B}$.
2. If $\Psi \vdash_{\text{GCAP1}} (\mathbb{C}, \Psi')$, then $\Psi \vdash_{\text{GCAP2}} [\![(\mathbb{C}, \Psi')]\!]$.
3. If $\Psi \vdash_{\text{GCAP1}} \mathbb{W}$, then $\Psi \vdash_{\text{GCAP2}} \mathbb{W}$.

Proof:

1. If $\mathtt{a} * \mathtt{blk}(\mathbb{B})$, By frame rule of CAP, we know $\mathtt{blk}(\mathbb{B})$ is always satisfied and not modified during the execution of $\mathbb{B}$. Thus the additional premise of is always satisfied. Then its' easy to get the conclusion by doing induction over the length of the code block.

2. With the help of 1, we see that if $\Psi \vdash_{\text{GCAP1}} (\mathbb{C}, \Psi')$, then for every $\mathtt{f} \in \mathsf{dom}(\Psi')$, we have $\mathtt{f} \in \mathsf{dom}(\mathbb{C})$ and $\Psi \vdash_{\text{GCAP2}} \{\Psi'(\mathtt{f})\}\mathtt{f} : \mathbb{C}(\mathtt{f})$. Then the result is obvious.

3. Directly derived from 2. ■

Finally, as promised, from Theorem 5.8 and the soundness of GCAP2 we directly see the soundness of GCAP1 (Theorem 4.2).

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).

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 24). Therefore certifying a boot loader is an important piece of a complete OS certification.

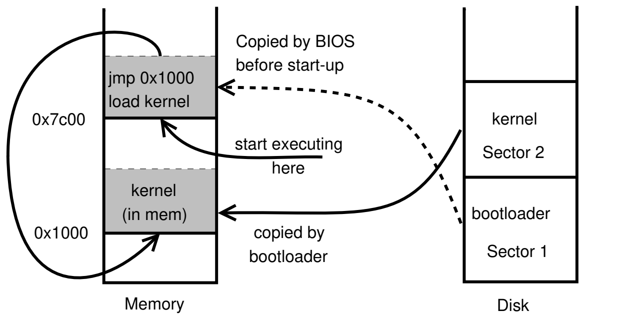

Figure 24. A typical boot loader

To show that we support a real boot loader, we have created one that runs on the Bochs simulator[19] and on a real x86 machine. The code (Fig 23) is very simple, it sets up the registers needed to 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.

6.2 Control Flow Modification

The fundamental operation unit of self-modification mechanism is the modification of single instructions. There are several possibilities of modifying an instruction. Opcode modification, which mutates an operation instruction (i.e. instruction that does not jump) into another operation instruction, has already been shown in Sec 5.2 as a typical example. This kind of code is relatively easier to deal with since it does not change the control flow of a program.

Another case which looks trickier is control flow modification. In this scenario, an operation instruction into a control transfer instruction, or the opposite way, and we can also modify a control transfer instruction into another.

Control flow modification is useful in patching of subroutine call address. And we give an example here that demonstrates control flow modification, and show how to verify it.

```
halt:   j    halt
main:   la   $9, end
        lw   $8, 0($9)
        addi $8, $8, -1
        addi $8, $8, -2
        sw   $8, 0($9)
end:    j    dead
dead:
```

The entry point of the program is the `main` line. At the first look, the code seems "deadly" because there is a jump to the `dead` line, which does not contain a valid instruction. But fortunately, before the program executes to that line, the instruction at it will be fetched

$\{\texttt{blk}(\texttt{main}: \mathsf{la}\ \$9,\texttt{end}; \mathbb{I}_1; \mathsf{j}\ \texttt{dead})\}$

```
main:   la   $9, end
        lw   $8, 0($9)
        addi $8, $8, -1
        addi $8, $8, -2
        sw   $8, 0($9)
        j    mid2
```

$\{(\lambda(\mathbb{M},\mathbb{R}).\langle\mathbb{R}(\$8)\rangle_4{=}\mathsf{Ec}(\mathsf{j}\ \texttt{mid2}) \wedge \mathbb{R}(\$9){=}\texttt{end}) * \texttt{blk}(\texttt{mid2}: \mathbb{I}_2; \mathsf{j}\ \texttt{mid2})\}$

```
mid2:   addi $8, $8, -2
        sw   $8, 0($9)
        j    mid1
```

$\{(\lambda(\mathbb{M},\mathbb{R}).\langle\mathbb{R}(\$8)\rangle_4{=}\mathsf{Ec}(\mathsf{j}\ \texttt{mid1}) \wedge \mathbb{R}(\$9){=}\texttt{end}) * \texttt{blk}(\texttt{mid1}: \mathbb{I}_1; \mathsf{j}\ \texttt{mid1})\}$

```
mid1:   lw   $8, 0($9)
        addi $8, $8, -1
        addi $8, $8, -2
        sw   $8, 0($9)
        j    halt
```

$\{\texttt{blk}(\texttt{halt}: \mathsf{j}\ \texttt{halt})\}$

```
halt:   j    halt
```

where

$$\mathbb{I}_1 = \mathsf{lw}\ \$8, 0(\$9); \mathsf{addi}\ \$8, \$8, -1; \mathbb{I}_2,$$
$$\mathbb{I}_2 = \mathsf{addi}\ \$8, \$8, -2; \mathsf{sw}\ \$8, 0(\$9)$$

Figure 25. Control flow modification

out, subtracted by 3 and stored back, so that the jump instruction on `end` now points to the second `addi` instruction. Then the program will continue executing from that line, until finally hit the `halt` line and loops forever.

Our specification for the program is described in Fig 25. There are four code blocks that needs to be certified. Then execution process is now very clear.

6.3 Runtime Code Generation and Code Optimization

Runtime code generation (also known as dynamic code generation), which dynamically creates executable code for later use during execution of a program[16], is probably the broadest usage of SMC. The code generated at runtime usually have better performance than static code in several aspects, because more information would be available at runtime. For example, if the value of a variable doesn't change during runtime, program can then generate specialized code with the variable as an immediate constant at runtime, which enables code optimizations such as pre-calculation and dead code elimination.

[28] gave plenty of utilization of run-time code generation. The certification of all those applications can be fit into our system. Here we show the certification of one in them. We demonstrate the runtime code generation version of a fundamental operation of matrix multiplication – vector dot product, which was also mentioned in [31].

Efficient matrix multiplication is important to digital signal processing and scientific computation problems. Sometimes the matrices might have certain characteristics such as sparseness(large number of zeros) or small integers. The use of runtime code generation allows these characteristics to be specialized on locally optimized code for each row of the matrix, based on the actual values. Since code for each row is specified once but used n times by the other matrix(one for each column), the performance improved easily surpasses the cost caused by the generation of code at runtime.

The code shown in Fig 26 (which includes the specification) will read the vector v_1 stored at `vec1`, generate specialized code at `gen` with elimination of multiplication by 0, and then run the code at `gen` to compute the dot product of v_1 and v_2, and store it into the memory address `result`. For vector $(22, 0, 25)$, the generated code would be as follows:

```
        .data   # Data declaration section
vec1:   .word 22, 0, 25
vec2:   .word 7, 429, 6
result: .word 0
tpl:    li   $2, 0          # template code
        lw   $13, 0($4)
        li   $12, 0
        mul  $12, $12, $13
        add  $2, $2, $12
        jr   $31

        .text   # Code section
          # {True}
main:   li   $4, 3          # vector size
        li   $8, 0          # counter
        la   $9, gen        # target address
        la   $11, tpl       # template address
        lw   $12, 0($11)
        sw   $12, 0($9)     # copy the 1st instr
        addi $9, $9, 4
          # {p(ℝ($8))}
loop:   beq  $8, $4, post
        li   $13, 4
        mul  $13, $13, $8
        lw   $10, vec1($13)
        beqz $10, next      # skip if zero
        lw   $12, 4($11)
        add  $12, $12, $13
        sw   $12, 0($9)
        lw   $12, 8($11)
        add  $12, $12, $10
        sw   $12, 4($9)
        lw   $12, 12($11)
        sw   $12, 8($9)
        lw   $12, 16($11)
        sw   $12, 12($9)
        addi $9, $9, 16
          # {p(ℝ($8)+1)}
next:   addi $8, $8, 1
        j    loop
          # {p(3)}
post:   lw   $12, 20($11)
        sw   $12, 0($9)
        la   $4, vec2
        jal  gen
          # {λ(𝕄,ℝ).ℝ($2)=v1·v2}
        sw   $2, result
        j    main
gen:
```

Figure 26. Vector Dot Product

$$\mathbb{I}_{k,u} \triangleq \begin{cases} \mathsf{lw}\ \$13, k(\$4); \mathsf{li}\ \$12, u; & \\ \quad \mathsf{mul}\ \$12, \$12, \$13; \mathsf{add}\ \$2, \$2, \$12, & \text{if } u > 0, \\ \emptyset, & \text{if } u = 0. \end{cases}$$

$$\mathsf{p}(k) \triangleq \lambda(\mathbb{M},\mathbb{R}).\mathbb{R}(\$4){=}3 \wedge k{\le}\mathbb{R}(\$4) \wedge \mathbb{R}(\$9){=}\texttt{gen}{+}4{+}16k \wedge$$
$$\texttt{blk}(\texttt{gen}: \mathsf{li}\ \$2, 0; \mathbb{I}_{0,\mathbb{M}(\texttt{vec1}+0)}; \ldots; \mathbb{I}_{k-1,\mathbb{M}(\texttt{vec1}+k-1)})\ (\mathbb{M},\mathbb{R})$$

Figure 27. Dot Product Assertion Macro

Since this program does not involve code-modification, we use GCAP1 to certify it. Each code block can be verified separately with the assertions we gave in Fig 26, Fig 27. The main part of the program can be directly linked and certified to form a well-formed code heap. On the other hand, the generated code, as in Fig 28, can

```
        # {ℝ($31) = back ∧ ℝ($4)=vec2}
gen:    li   $2, 0           # int gen(int *v)
        lw   $13, 0($4)      # {
        li   $12, 22         #   int res = 0;
        mul  $12, $12, $13   #   res += 22 * v[0];
        add  $2, $2, $12     #   res += 25 * v[2];
        lw   $13, 8($4)      #   return res;
        li   $12, 25         # }
        mul  $12, $12, $13
        add  $2, $2, $12
        jr   $31
```

Figure 28. Generated Code of Vector Dot Product

```
        # {True}
main:   jal  f
        # {λ(𝕄,ℝ).ℝ($8) = 42}
        move $2, $8
        j    halt

        # {(λ(𝕄,ℝ).ℝ($31)=main+4) ∗ blk(addr : jal f)
                                  ∗blk(main : jal f)}
f:      li   $8, 42
        lw   $9, -4($31)
        lw   $10, addr
        bne  $9, $10, halt
        jr   $31

        # {ℝ($2) = ℝ($8) = 42}
halt:   j    halt

addr:   jal  f
```

Figure 29. Runtime code-checking

be verified on its own. After these are all done, we use the - rule and the rule to merge the two parts together, which completes the whole verification. The modularity of our system guarantees that, the verification of the generated code (Fig 28 does not need any information about its “mother code”.

6.4 Runtime Code Checking

As we mentioned, the limitation of previous verification framework is not only about self-modifying code. They can also not deal with “self-reading code” that does not do any code modification.

The example below is a program where reading the code of itself helps the decision of the program executing. It is an interesting application of code-reading programs. As we know, there is a subroutine call instruction in most architectures (`jal` in MIPS or `call` in x86). The call instruction in essentially a composition of two operations: to first store the address of the next instruction to the return address register ($31 in MIPS), and then jump to the target address. When the subroutine finishes executing, hopefully, the return address register should point to the next instruction in the parent program, so that the program can continue execution normally. However, this is not guaranteed, because a vicious parent program can store a bad address into the return address register and jump to the subroutine. The code in Fig 29 shows how to avoid this kind of cheating by runtime code-checking mechanism.

The `main` program calls `f` as a subroutine. After the subroutine have done its job (storing 42 into register $8), and before return, it first checks whether the instruction previous to the return address register ($31) is indeed a jal (or call) instruction. If not so, it simply `halt`s the program; otherwise, return.

We verify it with GCAP1. The precondition of `f` asserts the return address, and both addr and main store the jal `f` instruction, so that the code block can check the equality of these two memory location. Note that the precondition of `main` does not specify the content of `addr` since this can be added later via the frame rule.

```
        .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 30. `fib.s`: Fibonacci number

```
       {blk(𝔹_1)}
      ⎧ main:   lw   $4, num
𝔹_1   ⎨         lw   $9, key
      ⎪         li   $8, 1
      ⎩         li   $2, 1
       {(λ(𝕄,ℝ).ℝ($4)=𝕄(num) ∧ ℝ($9)=Ec(addi $2,$2,0)∧
        ℝ($8)=k+1 ∧ ℝ($2)=fib(k+1)) ∗ blk(𝔹_{2,k})}
      ⎧ loop:   beq  $8, $4, halt
      ⎪         addi $8, $8, 1
      ⎪         add  $10, $9, $2
𝔹_2,k ⎨ key:    addi $2, $2, fib(k)
      ⎪         sw   $10, key
      ⎩         j    loop
       {(λ(𝕄,ℝ).ℝ($2)=fib(𝕄(num))) ∗ blk(𝔹_3)}
𝔹_3   { halt:   j    halt
```

Figure 32. `fib.s`: Code and specification

6.5 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 30. More specifically, `fib.s` will calculate $\mathsf{fib}(\mathbb{M}(\texttt{num}))$ 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 31 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 32. 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.

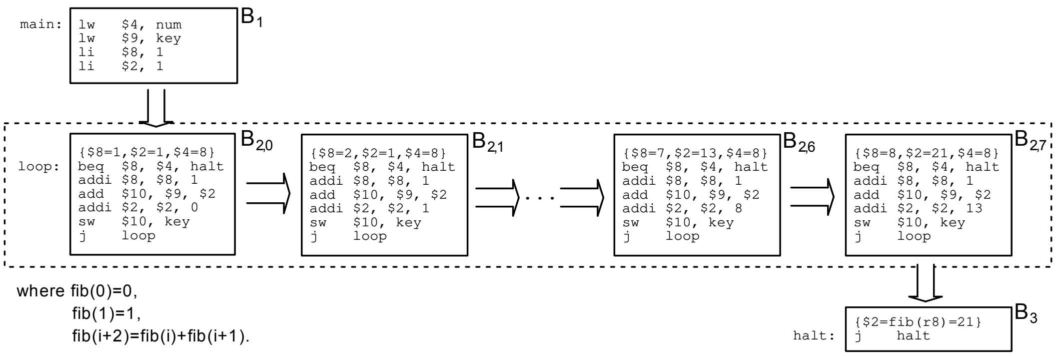

Figure 31. `fib.s`: Control flow

```
main:   la   $8, loop
        la   $9, new
        move $10, $9
loop:   lw   $11, 0($8)
        sw   $11, 0($9)
        addi $8, $8, 4
        addi $9, $9, 4
        bne  $8, $10, loop
        move $10, $9
new:
```

Figure 33. `selfgrow.s`: Self-growing Code

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 $\mathsf{addi}\ \$10,\$9,\$2$ 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 :

$$\Phi \triangleq \{\mathbb{B}_1 \rightsquigarrow \mathsf{a}_1,\ \mathbb{B}_{2k} \rightsquigarrow \mathsf{a}_{2k} \mid k{\in}\mathtt{Nat},\ \mathbb{B}_3 \rightsquigarrow \mathsf{a}_3\} \tag{6}$$

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 $\Phi \triangleq \lambda\mathbb{B}.\lambda\mathbb{S}.(\mathbb{B}{=}\mathbb{B}_1 \wedge \mathsf{a}_1\ \mathbb{S}) \vee (\exists k.\mathbb{B}{=}\mathbb{B}_{2k} \wedge \mathsf{a}_{2k}\ \mathbb{S}) \vee (\mathbb{B}{=}\mathbb{B}_3 \wedge \mathsf{a}_3\ \mathbb{S})$. One can easily see the equivalence between this definition and (6).

6.6 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 33. After initializing the registers, the code repeatedly duplicates itself and continue to execute the new copy , as Fig 34 shows .

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.

[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.

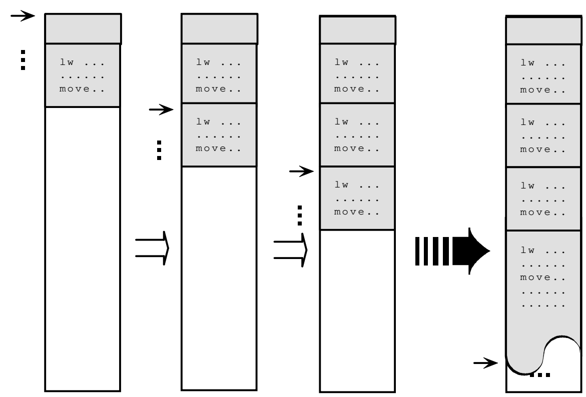

Figure 34. `selfgrow.s`: Self-growing process

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

$\mathbb{B}_0$:
```
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 35. `selfgrow.s`: Code and specification

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 35. 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.

6.7 Mutual-modifying Modules

In the circumstances of runtime code generation, the interaction between two modules is one-way: the “mother code” manipulates the “child code” but not the opposite. If both two modules operate

```
      {blk(𝔹1)}
𝔹1 {  main:   j    alter

      {(blk(𝔹1) ∗ blk(𝔹2))}
   (  alter:  lw   $8, main
𝔹2 {          li   $9, 0
   (          sw   $9, main
              j    main
      {(λ(𝕄,ℝ). ⟨ℝ($8)⟩4 = Ec(j alter)) ∗ blk(𝔹3)}
𝔹3 {  main:   nop
              sw   $8, alter

      {blk(𝔹4)}
𝔹4 {  alter:  j    alter
```

Figure 36. Specification - Mutual Modification

on each other, the situation becomes trickier. But with GCAP2, this can be solved without too much difference.

Here is a mini example showing what can happen under this circumstance.

```
main:   j    alter
        sw   $8, alter
alter:  lw   $8, main
        li   $9, 0
        sw   $9, main
        j    main
```

The `main` program will firstly jump to the `alter` block, and after it modified the instruction at `main` and transfer the control back, the `main` block will modify the `alter` instruction back, which makes the program loops forever.

The specification for this tangly is fairly simple, as shown in Fig 36. The actual number of code blocks that needs to be certified is 4, although there are only two starting addresses. Both the preconditions and the code body for different code blocks at the same starting address differs completely. For example, the first time the program enters `main`, it only requires $\mathbb{B}_1$ itself, while when the second time it enters, the code body changes to $\mathbb{B}_3$ and the precondition requires that register \$8 stores the correct value. In fact, by certifying this example, the control flow becomes much clearer to the readers.

6.8 Polymorphic Code

polymorphic code is code that mutates itself while keeps the algorithm equivalent. It is commonly used in the writing of computer virus and trojans who want to hide their presence for certain purpose. Since anti-virus software usually identify virus by recognizing particular byte patterns in the memory, the technique of polymorphic code has been a way to combat against this.

The following oscillating code is a simple demo example. Every time it executes through, content of the code would switch (between $\mathbb{B}_1$ and $\mathbb{B}_2$), but its function is always to add 42 to register \$2.

```
        # 𝔹0 =
main:   la   $10, body

        # 𝔹1 =                    # 𝔹2 =
body:   lw   $8, 12($10)          lw   $8, 12($10)
        lw   $9, 16($10)          lw   $9, 16($10)
        sw   $9, 12($10)          sw   $8, 16($10)
        j    dead                 addi $2, 21
        addi $2, 21               j    dead
        sw   $8, 16($10)          sw   $9, 12($10)
        lw   $9, 8($10)           lw   $9, 8($10)
        lw   $8, 20($10)          lw   $8, 20($10)
        sw   $9, 20($10)          sw   $9, 20($10)
        sw   $8, 8($10)           sw   $8, 8($10)
        j    body                 j    body
```

```
        {blk(𝔹0)}
main:   la   $10, body

        {blk(𝔹1)}                 {blk(𝔹2)}
body:   lw   $8, 12($10)          lw   $8, 12($10)
        lw   $9, 16($10)          lw   $9, 16($10)
        sw   $9, 12($10)          sw   $8, 16($10)
        addi $2, 21               addi $2, 21
        addi $2, 21               addi $2, 21
        sw   $8, 16($10)          sw   $9, 12($10)
        lw   $9, 8($10)           lw   $9, 8($10)
        lw   $8, 20($10)          lw   $8, 20($10)
        sw   $9, 20($10)          sw   $9, 20($10)
        sw   $8, 8($10)           sw   $8, 8($10)
        j    body                 j    body
```

Figure 37. Specification - Mutual Modification

The safety is not obvious as there is an illegal instruction j `dead` in our program. Since the code blocks that need to be certified overlap, we use GCAP2 to prove it. The specification is showed in Fig 37. Note the tiny difference between the code above and the code in Fig 37. This difference demonstrates the different understanding between "stored" code blocks and "executing" code blocks, and makes our verification work.

6.9 Code Obfuscation

The purpose of code obfuscation is to confuse debuggers. Although there are many other approaches that do code obfuscation without involving SMC such as control-flow based obfuscation, SMC is a much more effective way due to its hardness for understanding. By instrumenting a big amount of redundant, garbage SMC into normal code, plenty of memory reading and writing instructions will considerably confuse the disassembler. The method mentioned in [15] is a typical attempt. In PLDI'06's tutorial, Bjorn De Sutter also demonstrated their Diablo as a binary rewriting program to instrument self-modifying code for code-obfuscation.

To certify a SMC-obfuscated program, traditional verification techniques becomes extremely tough. At the same, the reasoning of this kind of code appears fairly simple under our framework. Take the following code as example:

```
   (  main:  la   $8, g
   (         lw   $9, 0($8)
   (         addi $10, $9, 4
   (         sw   $10, g
𝔹1 {         lw   $11, h
   (  g:     sw   $9, 0($8)
   (  h:     j    dead
   (         sw   $11, h
   (         j    main
```

The program does several weird memory loading and storing, and seems that there is a dead instruction, while actually it does nothing. The line `g` and `h` would change during the execution, and be changed back afterwards. Thus the code looks intact after one round of execution. If we remove the last line, the piece of code can be instrumented into regular program to fool the debugger.

In our GCAP2 system, such code can be verified with a single judgment, as simple as Fig 38 shows.

One observation is that this program can also be certified with the more traditional-like system GCAP1. But in contrast, since GCAP1, like traditional reasoning, doesn't allow a code block to mutate any of its own code, this program has to be divided into several smaller intact parts to certify, and makes each of the code blocks too small and too difficult to manipulate.

The development of our general code blocks solves the issue smoothly. Since a code block is not required to be self-immutable,

```
            {blk(𝔹₁)}
     main:  la   $8, g
            lw   $9, 0($8)
            addi $10, $9, 4
            sw   $10, g
            lw   $11, h
            sw   $9, 4($8)
            sw   $9, 0($8)
            sw   $11, h
            j    main
```

Figure 38. Specification - Obfuscation

```
       # {True}
main:    la   $8, pg
         la   $9, pgend
         li   $10, 0xffffffff
       # {(λ(𝕄,ℝ).(pg≤ℝ($8)<pgend)∧ℝ($9)=pgend∧
       #   ℝ($10)=0xffffffff)∗blk(𝔹_pg)}
xor1:    lw   $11, 0($8)
         xor  $11, $11, $10
         sw   $11, 0($8)
         addi $8, $8, 4
         blt  $8, $9, xor1
       # {True}
decr:    la   $8, pg
         la   $9, pgend
         la   $10, 0xffffffff
       # {(λ(𝕄,ℝ).(pg≤ℝ($8)<pgend)∧ℝ($9)=pgend∧ℝ($10)=
       #   0xffffffff∧(𝕄(pg),...,𝕄(pgend−1))=Ec(𝔹_pg)‾)}
xor2:    lw   $11, 0($8)
         xor  $11, $11, $10
         sw   $11, 0($8)
         addi $8, $8, 4
         blt  $8, $9, xor2
         j    pg
       # {ℝ($2) = 3}
halt:    j    halt

       # {True} 𝔹_pg =
pg:      li   $2, 1
         li   $3, 2
         add  $2, $2, $3
         j    halt
pgend:
```

Figure 39. `encrypt.s`: Code and specification

one code block can be as long as there's no control transferring . Observe that the sequence of executing code differs from the code block that actually stores in memory, see the specification in Fig 38.

6.10 Code Encryption and Code Compression

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 [29] with its specification is shown in Fig 39. The code block $\mathbb{B}_{pg}$ between the labels pg (inclusive) and pgend (exclusive) is the program that is going to be encrypted. In this example, $\mathbb{B}_{pg}$ simply calculates the sum of 1 and 2 and stores the result 3 into the register $2, as the precondition of `halt` indicates.

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

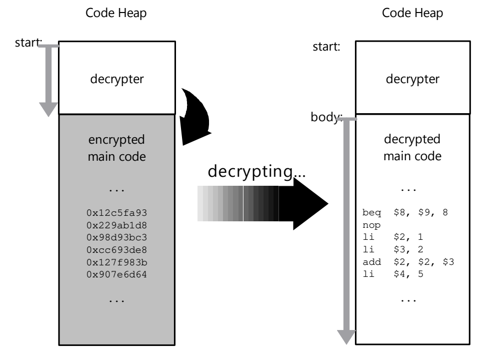

Figure 40. The execution of runtime code decryption

$\mathbb{B}_{pg}$ is properly stored, while `xor2` on the contrary needs to make sure that the flip of $\mathbb{B}_{pg}$ is properly stored (as 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 40.

By making use of parametric code, It is possible to certify this encrypt-decrypter even without knowledge of the content of $\mathbb{B}_{pg}$. That is, we abstract $\mathbb{B}_{pg}$ out as a parameter, and prove the general property of the main code $\mathbb{C}$, no matter what $\mathbb{B}_{pg}$ is actually used:

$$\forall \mathbb{B}_{pg}.(\{\mathtt{halt} \rightsquigarrow \mathtt{a}\} \vdash \{\mathtt{True}\}\mathbb{B}_{pg}) \longrightarrow \exists \Psi.(\Psi(\mathtt{main})=\mathtt{blk}(\mathbb{C}) * \mathtt{blk}(\mathbb{B}_{pg})) \wedge (\Psi(\mathtt{pg}) \Rightarrow \mathtt{blk}(\mathbb{B}_{pg})) \wedge (\Psi(\mathtt{halt}) \Rightarrow \mathtt{a}) \wedge (\Psi \vdash (\mathbb{C} \cup \mathbb{B}_{pg}, \Psi))\}$$

This statement is expressing the following fact : given any code block $\mathbb{B}_{pg}$, 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 $\mathbb{B}_{pg}$ 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 $\mathbb{B}_{pg}$ (meaning that they are both well-formed with respect to the same precondition and specification).

Runtime code decompression behaves similar to runtime code decryption, except that the code is compressed before runtime rather than encrypted. Given this fact, the verification of it would be not too much different from runtime code decryption.

6.11 Shellcoding

Shellcode is usually a piece of machine code that does certain task, especially opens a shell. It is often used by hackers to control remote computer through buffer overflow exploits. Many shellcode do self-modifying. Sometimes shellcode will do partial code encryption. For example, since most shellcode is input as strings, and might encounter some character filtering, one needs to let the original code appear no filtered bytes but modifies it back during executing. In this way, one can even construct alphanumeric code that completely consists of alphanumeric characters. The self-modifying techniques in shellcoding have nothing new than previous examples and can be certified without surprise.

7. Implementation

We use the Coq proof assistant [32], which comes with an expressive underlying logic, to implement our full system, including the formal construction of GTM, $\mathcal{M}_{\mathrm{MIPS}}$, $\mathcal{M}_{\mathrm{x86}}$, GCAP0, GCAP1, and GCAP2, as well as the proof of soundness and related properties of all the three systems. The assertion language as well as the basic properties of separation logic are also implemented and proved.

We have certified several typical examples, including a complete OS boot loader written in x86 for the demonstration of GCAP1 and the Fibonacci example in MIPS for the use of GCAP2.

The system can be immediately used for more realistic and theoretical applications.

8. Related Work and Conclusion

Previous assembly code certification systems (e.g., TAL [23], FPCC [1, 10], and CAP [33, 25]) all treat code separately from data memory, so that only immutable code is supported. Appel *et al* [2, 22] 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, 31] 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 [24]—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 *Proceedings of the 2002 ACM Conference on Programming Language Design and Implementation*, pages 95–105, New York, NY, 2002. ACM Press.

[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] D. Keppel, S. J. Eggers, and R. R. Henry. A case for runtime code generation. Technical Report UWCSE 91-11-04, University of Washington, November 1991.

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

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

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

[20] 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.

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

[22] 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.

[23] 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.

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

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

[26] 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.

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

[28] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. 'C and tcc: a language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, 1999.

[29] Ralph. Basics of SMC. `http://web.archive.org/web/`

20010425070215/awc.rejects.net/files/text/smc.txt, 2000.

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

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

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

[33] 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.