

# Computer Sciences Department

**MCDASH: Refinement-Based Property Verification for Machine  
Code**

Akash Lal  
Junghee Lim  
Thomas Reps

Technical Report #1659

June 2009



# MCDASH: Refinement-Based Property Verification for Machine Code<sup>\*</sup>

Akash Lal    Junghee Lim    Thomas Reps

University of Wisconsin

{akash, junghee, reps}@cs.wisc.edu

## Abstract

This paper presents MCDASH, a refinement-based model checker for machine code. While model checkers such as SLAM, BLAST, and DASH have each made significant contributions in the field of verification/bug-detection, their use has been restricted to programs for which source code is available. This paper discusses several challenges that arise when working with machine code, and explains how they are addressed in MCDASH. Unlike previous model checkers, MCDASH does not require the usual preprocessing steps of (a) building control-flow graphs, and (b) performing points-to analysis (or alias analysis); nor does MCDASH require type information to be supplied. The paper also describes how we extended MCDASH to check properties of self-modifying code.

MCDASH is built using language-independent meta-tools that generate the implementations of the required analysis components from descriptions of an instruction set's syntax and semantics. It has been instantiated for Intel x86 and PowerPC.

## 1. Introduction

Recent research in programming languages, software engineering, and computer security has led to new kinds of tools for analyzing programs for bugs and security vulnerabilities [33, 25, 36, 19, 13, 8, 5, 11, 26, 16, 1]. In these tools, program analysis is used to determine a conservative answer to the question “Can the program reach a bad state?” Many impressive results have been achieved, and some of this work has already been transitioned to commercial products [8, 4, 15, 12].

However, these tools all focus on analyzing *source code*. Unfortunately, most programs that an individual user will install on his computer, and many commercial off-the-shelf programs that a company will purchase, are delivered as machine code. If an individual or company wishes to vet such programs for bugs, security vulnerabilities, or malicious code (e.g., back doors, time bombs, or logic bombs) the availability of good source-code-analysis products is irrelevant. For instance, because device-driver developers rarely make their source code available, we can only *trust* that they have run Microsoft's Static Driver Verifier (SDV) [4] on their code and fixed the bugs that were found; we are not in a position to run SDV ourselves because it does not work on machine code.

<sup>\*</sup>The research was supported by NSF under grants CCF-0540955, CCF-0524051, and CCF-0810053, and by AFRL under contract FA8750-06-C-0249.

Although establishing execution properties at the machine-code level is a challenging task, the problem of analyzing machine code has been receiving increased attention [28, 17, 2, 9, 22, 7]. Moreover, it can be a useful complement to source-code analysis, even when source code is available:

- The compilation from source code to machine code can introduce subtle but important differences between what a programmer intended and what is actually executed by the processor. However, source-code analyses are blind to the choices made by the compiler. The effects of compilation can only be detected by examining the machine code emitted by the compiler.
- In addition to the machine code that a programmer creates by compiling his source code, additional machine code is linked in either statically or dynamically from libraries. Often the source code for these libraries is not available, and thus cannot be analyzed by a source-code-analysis tool. However, a machine-code-analysis tool can analyze a library's machine code.

For these reasons, we have developed a model checker for machine code, called MCDASH. The work on MCDASH addresses the problem of creating a model checker that is (i) capable of verifying properties of machine-code programs, and (ii) can be retargeted easily to different instruction sets automatically. In particular, MCDASH is built using language-independent meta-tools that generate the implementations of the required analysis components from descriptions of an instruction set's syntax and semantics. To date, versions of MCDASH have been instantiated for the Intel x86 and PowerPC instruction sets.

Previous model checkers, such as SLAM [5] (the core component of SDV), BLAST [26], MAGIC [10], and DASH [6], have each made significant contributions in the field of verification/bug-detection; however, their use has been restricted to source-code programs written C. Although C is already quite low-level, there are a number of issues that arise in the analysis of machine code that are not handled by the model checkers mentioned above.

**Pointers and Types:** SLAM, BLAST, and MAGIC use points-to analysis as a preprocessing step before starting the verification process proper. They rely on the points-to analysis to be efficient and reasonably precise to get good overall performance. Current versions of these tools use a flow-insensitive (and possibly field-sensitive) points-to analysis that makes unsound assumptions about pointer arithmetic—they either ignore pointer arithmetic altogether (SLAM) or assume that the result of an arithmetic operation on a pointer is always contained inside the object that the pointer pointed to originally (BLAST and MAGIC).

The latter approach amounts to making an unchecked assumption that the program is ANSI C compliant. The consequence is that such model checkers do not account for behaviors that are allowed by some compilers (e.g., arithmetic is performed on pointers that are subsequently used for indirect function calls; pointers move off the ends of structs or arrays, and are subsequently dereferenced; etc.) There can be good reasons why a program uses such features—e.g., as a way to simulate subclassing in C [35]—but they can also lead to bugs and security vulnerabilities.

Existing model checkers also typically depend on some form of type information, e.g., to distinguish array variables from scalar variables, or to ensure that dereferences of an address-valued quantity are compatible with the type of the objects to which it refers.

For machine-code programs, making such assumptions is unreasonable: (i) An access on a local variable is compiled to an instruction operand that dereferences a computed address. For instance, if local variable  $x$  is at offset  $-12$  from the activation record's frame pointer (register  $\text{ebp}$ ), an access on  $x$  would typically be turned into an operand  $[\text{ebp}-12]$ , which dereferences the computed address  $\text{ebp}-12$ . (ii) Type information may not be available for the objects to which an address-valued quantity refers.

DASH does not require a preprocessing step of points-to analysis, and it does handle pointer arithmetic to some extent. However, it still requires type information to distinguish pointer variables from scalar variables. In addition, equality and disequality constraints between pointer values are used to identify an *aliasing condition* relevant to a specific property in a specific state. The use of such aliasing conditions is central to DASH's ability to perform verification in the absence of a separate points-to analysis: the aliasing conditions are acquired "on-the-fly"—during the course of verification—instead of ahead of time.

In machine code, int-valued quantities and address-valued quantities are indistinguishable at runtime, and arithmetic on addresses is used extensively. This makes it challenging to define the appropriate notion of "aliasing condition" for use in MCDASH.

**Byte-Addressable Memory:** In x86 machine code, memory is byte-addressable, and a sound analysis must be able to handle non-aligned addresses.

**Variables and Arrays:** For source-level tools, an access to a stack-allocated variable is not modeled as a dereference of a memory address. Programs in which the property of interest can be proven while reasoning about only stack variables provide easy cases for source-level tools. In machine code, however, such programs are not as easy because every access on a stack-allocated variable is performed via a memory dereference.

One shortcoming of DASH vis à vis machine code is that it treats array accesses unsoundly (the way SLAM does). This allows the DASH tool to avoid using the theory of arrays inside its theorem prover (which improves the tool's scalability). However, at the machine-code level, memory looks like one large array.

For both of these reasons, it is challenging for a machine-code model checker to achieve the same degree of scalability as a source-code model checker.

**Control Flow:** Most front ends for processing a language's source code provide a reasonably accurate description of a program's control flow (using points-to-analysis results to supply missing information about the callees of an indirect function call).

For machine-code analysis, there are several reasons why recovering control flow is challenging.

- The branch condition is often not explicit: many instruction sets provide separate instructions for (i) setting flags (based on some condition that is tested) and (ii) subsequent branching according to the values held by one or more flags.
- It is often difficult to identify the targets of indirect jumps and indirect function calls [3].

**MCDASH.** MCDASH is based on the DASH algorithm of Beckman et al. [6]. DASH uses concrete testing along with symbolic reasoning to find either a test input that definitely causes a (bad) target state to be reached, or a proof that the bad state can never be reached. (The third possibility is that DASH may fail to terminate.)

In the MCDASH implementation, we use a technique due to Lim et al. [30] to generate automatically some of the key primitives from a description of the concrete semantics of an instruction set. This creates (a) an emulator for running tests, (b) a primitive for

performing symbolic execution, and (c) a primitive for performing weakest-liberal-precondition (WLP).

This provided a starting point for our work, but to create MCDASH we still had to address all of the challenges discussed above. In doing so, we restricted ourselves to use only language-independent techniques. Consequently, the overall system acts as a "Yacc-like" tool for creating versions of MCDASH for different machine-code instruction sets: given a description of an instruction set, a MCDASH-based model checker is generated automatically. This infrastructure has been used to generate two such model checkers: MCDASH/x86 and MCDASH/PowerPC.

For a given instruction set, we can actually create three different kinds of MCDASH model checkers:

**MCDASH-ICFG:** This version is useful in contexts in which it is possible to create an accurate interprocedural control-flow graph (ICFG)—that is, when source code, a cooperative compiler, and/or symbol-table/debugging information are available. In particular, MCDASH-ICFG uses the ICFG to build its initial abstraction of the program's state space. (In essence, it abstracts states based on the value of the program counter.)

**MCDASH- $\overline{\text{ICFG}}$ :** Because it is not possible, in general, to build an accurate ICFG for machine-code programs without assistance from the compiler, MCDASH- $\overline{\text{ICFG}}$  uses an initial abstraction of the state space that is coarser than the ICFG. It consists of three abstract states defined by the predicates "PC = start", "PC = target", and "PC  $\neq$  start  $\wedge$  PC  $\neq$  target" (where "PC" denotes the program counter). MCDASH- $\overline{\text{ICFG}}$  refines this abstraction during the course of state-space exploration.

**MCDASH-SMC:** This version is capable of verifying (or detecting flaws in) self-modifying code (SMC). (Self-modifying code is used in runtime code generation, code-encryption schemes, and OS boot loading. It is also used in malware.)

The work described in the paper makes the following contributions:

1. MCDASH extends DASH in several ways.
  - (a) MCDASH does not require any preprocessing analysis, such as points-to analysis, alias analysis, and control-flow analysis; nor does it require information that identifies the program's variables or their types.
  - (b) We developed a language-independent way for MCDASH to identify the aliasing condition relevant to a specific property in a specific state.
  - (c) We developed a way to speed up MCDASH—*without impacting soundness*—using a technique from concolic execution: some symbolic values are replaced with concrete values taken from the concrete state. This reduces the size and complexity of the formulas sent to the theorem prover.
  - (d) We introduced several optimizations to regain some of the scalability lost by moving to a low-level language.
2. We extended MCDASH to deal with self-modifying code (SMC). This is not possible with most other model checkers because they make a premature—and, in general, unsound—commitment to the ICFG as an abstraction of a program's state space. As far as we know, MCDASH-SMC is the first model checker to address verifying (or detecting flaws in) SMC.

**Organization.** The remainder of the paper is organized as follows: §2 reviews the DASH algorithm for model checking source-code. §3 describes MCDASH-ICFG, our simplest approach to extending the DASH algorithm to work on machine code. §4 describes MCDASH- $\overline{\text{ICFG}}$ , which can be used when it is not possible to start with an accurate ICFG of a machine-code program. §5 presents MCDASH-SMC, which addresses self-modifying code. §6 describes how a language-independent MCDASH implementation was created. §7 presents some experiments carried out with MCDASH/x86. §8 discusses related work.

---

**Algorithm 1** Single DASH Iteration

---

**Input:** An abstract graph  $G$  with *start* and *target* nodes.

**Input:** A set of concrete traces  $T$ .

```
1: if target has a witness in  $T$  then
2:   return reachable
3: end if
4: Find a path  $\tau$  in  $G$  from start to target.
5: if no path exists then
6:   return not reachable
7: end if
8: Find the last edge  $(n, m)$  of  $\tau$  such that  $n$  has a witness  $c$  in  $T$ .
9: Let  $I$  be the instruction on edge  $(n, m)$ .
10: Symbolically execute the concrete trace to  $c$  and then  $I$ .
11: Let  $S$  be the symbolic state obtained.
12: if  $S$  is feasible then
13:   Find program input from  $S$ , run test, and add trace to  $T$ .
14: else
15:   Refine  $G$  at node  $n$ .
16: end if
```

---

## 2. Background on DASH

This section provides an overview of how the DASH algorithm [6] operates on source code. Given a program and a target label (a particular control location in the program), DASH either returns a test case whose execution leads to the target, or a proof that the target is unreachable (or DASH does not terminate).

While DASH is running, it maintains an approximation of the program’s state space. The approximation is represented as a graph with edges labeled with program statements or program conditions, and nodes labeled with formulas. We call such a graph an *abstract graph*. One of the nodes in the graph is designated to be the start node (where program execution starts) and another node is designated as the target (representing the target label). DASH also stores a set of concrete traces  $T$  that are obtained from running tests. A concrete state in  $T$  is said to be a *witness* for a node  $n$  in the abstract graph if it satisfies the formula that labels node  $n$ .

DASH proceeds iteratively. During each iteration, it either runs a test (in an attempt to reach the target) or refines the abstract graph by splitting nodes and removing certain edges (in an attempt to prove that the target is not reachable). If the graph has no path from start to target, then DASH has proved that target is unreachable, and the abstract graph serves as the proof. An informal description of a single DASH iteration is shown in Alg. 1.

We will explain the algorithm using the program shown in Fig. 1(a) as an example, and describe how DASH proves or disproves the reachability of each of the labels L1, L2, and L3. First, suppose that the target is L1. DASH starts with an abstract graph  $G$  that is the control-flow graph (CFG) of `foo`, shown in Fig. 1(b), and all nodes are labeled with the formula *true*. It initializes  $T$  by running a random test. For our example, the only input to the program is the value of  $x$ . DASH chooses a random value for  $x$ , say 10, and runs a test. This produces a trace of concrete states that witness nodes of  $G$ . The presence of a witness for a node of  $G$  is shown in Fig. 1(b) as an “ $\times$ ” inside the node.

In the first iteration, step 1 does not find a witness for L1. Next, step 4 finds the path  $\tau = \text{foo\_start} \xrightarrow{y=x+1} n_1 \xrightarrow{y==1} \text{L1}$ . In steps 8 and 9, the node  $n$  is  $n_1$ , and  $I$  is `assume(y == 1)`. Step 10 performs symbolic execution.

A symbolic state has two components: a *path constraint* and a *symbolic map*. The initial symbolic state has path constraint *true* and a symbolic map that assigns a symbolic constant to the input:  $[x \mapsto x_0]$ . Symbolic execution proceeds by building formulas and expressions over the symbolic constants. The execution of the

assignment  $y = x + 1$  does not change the path constraint, but changes the symbolic map to  $[x \mapsto x_0, y \mapsto x_0 + 1]$ . The next statement gathers up a path constraint: it equates the current value of  $y$  with 1, leading to the constraint  $x_0 + 1 == 1$ , which is conjoined to the existing path constraint.

Thus, in step 11, the symbolic state has path constraint  $x_0 + 1 == 1$  and map  $[x \mapsto x_0, y \mapsto x_0 + 1]$ . A symbolic state is feasible if and only if its path constraint is feasible. In this case, it is feasible under the assignment  $x_0 = 0$ . This provides a new test case, and `foo` is executed with  $x$  initialized to 0. During the second iteration, step 1 finds that L1 has a witness: the test reaches L1.

Now suppose that the target is L2. As before, DASH starts with  $G$  as the CFG of `foo` and runs a random test with, say,  $x$  assigned to 10 again (so that Fig. 1(b) still describes the initial situation). In the first iteration,  $\tau$  is the (unique) path from `foo_start` to L2. In steps 8 and 9, DASH considers the frontier  $(n_3, \text{assume}(z == 0), \text{L2})$ . Symbolic execution yields the path constraint  $(x_0 + 1 \neq 1 \wedge 2x_0 = 0)$ , which is unsatisfiable (assuming integer arithmetic, to keep the discussion simple). In this case, DASH refines  $G$ . Next, we explain how DASH carries out its refinement, in general, and will then continue with our example.

The triple  $(n, I, m)$ , where node  $n$  and instruction  $I$  are the ones chosen in steps 8 and 9, and  $m$  is the successor node of  $I$ , is called the *frontier*: node  $n$  is the last place (along the currently chosen path) at which a concrete witness has been seen, and DASH tries to push a test beyond  $I$  in the hope that it might lead to the target. When this is not possible, the abstract graph  $G$  is refined by splitting node  $n$  into  $n'$  and  $n''$ , as shown in Fig. 1(c). The refinement operation allows some *non-connectivity* information to be represented in  $G$ ; in particular, refinement is performed in such a way that the refined graph records that  $n'$  is not connected to  $m$  (see Fig. 1(c)).

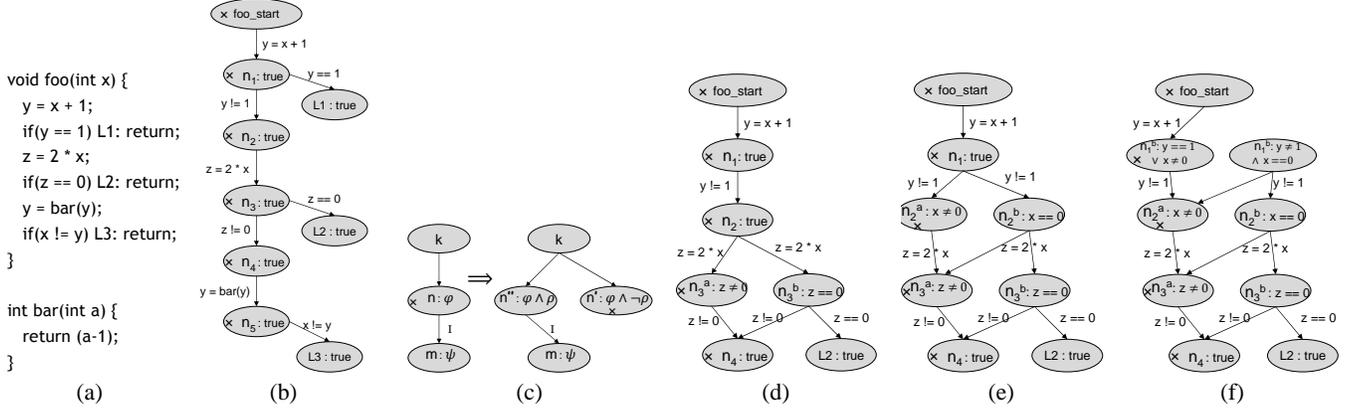
Let  $\psi$  be the formula that labels  $m$ ,  $c$  be the concrete witness of  $n$ , and  $S_n$  be the symbolic map obtained from the symbolic execution of  $\tau$  up to  $n$ . DASH chooses a formula  $\rho$ , called the *refinement predicate*, and splits node  $n$  into  $n'$  and  $n''$  to distinguish the cases when  $n$  is reached with a concrete state that satisfies  $\rho$  ( $n''$ ) and when it is reached with a state that satisfies  $\neg\rho$  ( $n'$ ). This predicate is chosen such that

- (i) no state that satisfies  $\neg\rho$  can lead to a state that satisfies  $\psi$  after the execution of  $I$ , and
- (ii) the symbolic map  $S_n$  satisfies  $\neg\rho$ .

The first condition ensures that the edge from  $n'$  to  $m$  can be removed, and the second condition rules out the possibility of extending the current path along  $I$  (forcing the search to explore different paths). It also ensures that  $c$  is now a witness for  $n'$  and not  $n''$  (because  $c$  satisfies  $S_n$ )—and thus the frontier during the next iteration must be different. One possibility for the refinement predicate is to choose the weakest liberal precondition (WLP) of  $\psi$  with respect to  $I$ . Other possibilities are discussed below.

Returning to the example of how node  $n_3$  is refined across the frontier  $(n_3, \text{assume}(z == 0), \text{L2})$ , DASH chooses the refinement predicate  $(z == 0) \wedge \text{true}$ , which simplifies to  $z == 0$ . This leads to the abstract graph shown in Fig. 1(d). (The concrete witnesses are again shown as  $\times$ 's.)

This case, when  $I$  is of the form `assume( $\varphi$ )`, is one in which DASH chooses a refinement predicate other than  $\rho_1 \stackrel{\text{def}}{=} \text{WLP}(I, \psi)$  (where  $\psi$  is the formula that labels  $m$ ). The reason is that  $\text{WLP}(\text{assume}(\varphi), \psi)$  equals  $\varphi \Rightarrow \psi$  [29]. For instance, in the example above,  $\rho_1$  would be  $\text{WLP}(\text{assume}(z == 0), \text{true}) = ((z == 0) \Rightarrow \text{true})$ , which simplifies to *true*. However, in keeping with condition (i) above—i.e., states that cannot satisfy  $\psi$  after the execution of  $I$  should satisfy  $\neg\rho$ —we use the stronger refinement predicate  $\rho_2 \stackrel{\text{def}}{=} (\varphi \wedge \psi)$ . This shifts all states that satisfy  $\neg\varphi$  to the refined node labeled with  $\neg\rho_2$  (e.g.,  $n_3^g$  in Fig. 1(d)). For instance,



**Figure 1.** (a) An example program. (b) Initial abstract graph created by DASH, with witnesses shown using “×”. (c) General DASH refinement. (d)–(f) The abstract graphs after different iterations of DASH. (To reduce clutter, nodes that cannot reach L2 are omitted.)

in Fig. 1(d) all states that satisfy  $z \neq 0$  are excluded from node  $n_3^b$ . Because  $n_3^a$  is not connected to L2, refinement via  $\rho_2 = (\varphi \wedge \psi)$  allows the refined abstract graph to represent more information about non-connectivity of states than it would have via  $\rho_1 = (\varphi \Rightarrow \psi)$ .

In the next iteration,  $\tau$  is chosen to be  $[\text{foo\_start}, n_1, n_2, n_3^b, \text{L2}]$ , and the frontier is at node  $n_2$ . While performing symbolic execution, the formulas that label the nodes are picked up in the path constraint: i.e., if the current symbolic map is  $S$  and  $\phi$  is the formula on the current node, then  $\phi$  is evaluated (similar to a branch condition) and conjoined in the path constraint. In our example, the formula that labels  $n_3^b$  will be picked up, leading to the same path constraint as before (which means that the current path cannot produce a concrete witness for  $n_3^b$ ). Refinement is performed at  $n_2$ , leading to the graph shown in Fig. 1(e). This continues, finally leading to the graph shown in Fig. 1(f). (In the last iteration, the refinement predicate turns out to be *false*, and nodes labeled with *false* are deleted from the abstract graph.) This graph proves that L2 is unreachable.

**Refinement Predicate.** In the presence of pointers, choosing the right refinement predicate is the key. Suppose that the frontier  $(n, I, m)$  has statement  $I = *p = 5$  and that the formula on  $m$  is  $\psi = (x + y == 10)$ . Then  $\text{WLP}(\psi, I)$  is

$$\begin{aligned}
& p == \&x \quad \wedge \quad p == \&y \quad \wedge \quad (5 + 5 == 10) \\
\vee & p == \&x \quad \wedge \quad p \neq \&y \quad \wedge \quad (5 + y == 10) \\
\vee & p \neq \&x \quad \wedge \quad p == \&y \quad \wedge \quad (x + 5 == 10) \\
\vee & p \neq \&x \quad \wedge \quad p \neq \&y \quad \wedge \quad (x + y == 10)
\end{aligned}$$

This formula has four disjuncts, each for a different *aliasing condition*. DASH defines an aliasing condition  $\alpha$  as a conjunction of equality and disequality constraints between addresses that are written to when executing the program statement (i.e.,  $p$ ) and ones that are used in the formula (i.e.,  $\&x$  and  $\&y$ ).

In general, if the statement writes to just one address but the formula has  $n$  addresses, then there are  $2^n$  possible aliasing conditions. The key insight that allows DASH to operate efficiently in the presence of pointers is that it chooses the refinement predicate based on aliasing conditions that actually arise in the program execution. It looks at  $S_n$ , which represents a collection of concrete states that actually arise during program execution, and derives  $\alpha$  from it. The intuition behind this approach is that one does not expect too many aliasing conditions to arise at a particular point in the program. Thus, considering them lazily makes the overall process efficient.

In the example above, suppose that  $S_n$  is  $[p \mapsto \&x, \dots]$ , and the addresses of  $x$  and  $y$  are distinct. Then  $\alpha = (p == \&x \wedge p \neq \&y)$ .

DASH chooses the refinement predicate  $\text{WLP}_\alpha(I, \psi) = \neg\alpha \vee (\alpha \wedge \text{WLP}(I, \psi))$ . The latter term  $(\alpha \wedge \text{WLP}(I, \psi))$  selects only one conjunct from WLP out the exponentially many that it may have. This allows DASH to avoid the exponential blowup. One can verify that  $\text{WLP}_\alpha$  is a valid refinement predicate.

**Interprocedural Analysis.** Now suppose that the target is L3. DASH operates as before, except when the frontier is a call statement. In its first iteration, DASH splits node  $n_5$  using the refinement predicate  $x \neq y$ . In the next iteration, the frontier is the call to procedure *bar*.

At a frontier, DASH essentially needs to determine whether a test could go beyond the frontier. Thus, in this case, it needs to find out if the execution of *bar* can produce a concrete state that satisfies the formula  $\psi = (x \neq y)$ . It does this by recursively calling itself on procedure *bar*: the target is set to be the WLP of  $\psi$  across the assignment of the return value of *bar* to  $y$ , which is  $\psi' = (x \neq a)$ . This is done by splitting the exit node of *bar* into two nodes, one labeled with  $\psi'$  and the other with  $\neg\psi'$ . The former node is set as the target node. The constraints on parameters of *bar* are obtained from the symbolic state  $S_{n_4}$  that is obtained from symbolically executing the concrete trace up to node  $n_4$ .

Thus, the recursive call to DASH is obligated to start from a state  $[a \mapsto x_0 + 1, x \mapsto x_0]$  with  $(x_0 + 1 \neq 1 \wedge 2x_0 \neq 0)$ , and must prove or disprove  $\psi'$  at the end of *bar*. If this call to DASH returns a test case, then the frontier inside *foo* can be extended using the same test; otherwise, it proves that  $\psi'$  cannot be reached, in which case a refinement is performed at the frontier in *foo*. The refinement predicate is obtained from the proof that is returned by DASH (see [6] for details). In our example, the refinement predicate would be  $(x \neq y + 1)$ .

During interprocedural analysis, additional care has to be taken because there are now two targets: the split exit node labeled with  $\psi'$ , as well as the global analysis target. On a given iteration, DASH may use a path in the abstract graph  $G$  to either target.

**Checking Safety Properties.** We have only described DASH for when the goal is to test the reachability of a given program location. However, DASH can handle general safety properties as well: if one wants to verify if the formula  $\varphi$  is ever violated during the execution of the program, then all nodes of the initial abstract graph are split into two, one labeled with  $\varphi$  and the other labeled with  $\neg\varphi$ . All nodes labeled with  $\varphi$  are treated as the target node. MCDASH is also able to do the same, but we limit the discussion to properties that check the reachability of a single program location.

### 3. MCDASH-ICFG

We now describe how to extend the DASH algorithm to work on machine code. The starting point is Alg. 1, which has four requirements:

1. access to an interprocedural control-flow graph (ICFG), which is used to build the initial abstract graph
2. a method to perform concrete execution of the program for running tests
3. a method to perform symbolic execution of a given program path (a sequence of program statements)
4. a method to compute  $WLP_\alpha$ , which is used when refining the abstract graph.

In our first version of MCDASH, called MCDASH-ICFG, we assume that the ICFG of the machine-code program is provided to us. In particular, the MCDASH-ICFG implementation uses the CodeSurfer/x86 front end to build CFGs. (This assumption is dropped in §4 and §5.)

#### 3.1 A Language-Independent Approach to $WLP_\alpha$

Lim et al. showed how to create primitives for concrete execution and symbolic execution of machine code [30]. Their work provides items 2 and 3 listed above. They also showed how to create a primitive for weakest liberal precondition (WLP), but that primitive causes the predicates that label the nodes of the abstract graph to explode.

In this section, we describe our language-independent approach to identifying “aliasing condition”  $\alpha$ , as well as the  $WLP_\alpha$  primitive that goes along with it.

##### 3.1.1 $\alpha$ and $WLP_\alpha$

As mentioned in §1, there are two challenges in defining an appropriate notion of *aliasing condition* for use with machine code.

- int-valued quantities and address-valued quantities are indistinguishable at runtime, and
- arithmetic on addresses is used extensively.

Suppose that the frontier is  $(n, I, m)$ ,  $\psi$  is the formula on  $m$ , and  $S$  is the symbolic state for the path up to  $n$ . For source code, aliasing condition  $\alpha$  can be derived by looking at the relationship, in  $S$ , between the addresses written to by instruction  $I$  and the ones used in  $\psi$  [6]. However, this way of computing  $\alpha$  is *language-dependent* because the semantics of the language of instructions must be incorporated into the algorithm, to determine “the addresses written to by instruction  $I$ ”.

In contrast, we developed an alternative, language-independent approach both to identifying  $\alpha$  and computing  $WLP_\alpha$ . For the moment, to simplify the discussion, suppose that a concrete machine-code state is represented using two maps  $M : INT \rightarrow INT$  and  $R : REG \rightarrow INT$ . Map  $M$  represents memory, and map  $R$  represents the values of machine registers. (A more realistic definition of memory is considered in §3.1.2.)

We use the standard theory of arrays to describe updates and accesses on maps, e.g.,  $update(M, k, d)$  denotes the map  $M$  with index  $k$  updated with the value  $d$ , and  $access(M, k)$  is the value stored at index  $k$  in  $M$ . We also use the standard axiom from the theory of arrays:

$$access(update(M, k_1, d), k_2) = ite(k_1 = k_2, d, access(M, k_2)), \quad (1)$$

where *ite* is an *if-then-else* term. We use the notation  $R(r)$  as a shorthand for  $access(R, r)$ .

Consider the following machine-code example, which is similar to the source-code example discussed in §2. Suppose that  $I$  is “mov [eax], 5” (which corresponds to  $*eax = 5$  in source-code notation), and  $\psi$  is  $(M(R(\text{ebp}) - 8) + M(R(\text{ebp}) - 12) = 10)$ .

Also, suppose that under the symbolic state  $S$ ,  $R(\text{eax})$  equals  $R(\text{ebp}) - 8$ .<sup>1</sup>

First, we symbolically execute  $I$  starting from the identity symbolic state  $S_{id} = [M \mapsto M_0, R \mapsto R_0]$ . This results in the symbolic state  $S' = [M \mapsto update(M_0, R_0(\text{eax}), 5), R \mapsto R_0]$ . Next, we evaluate  $\psi$  under  $S'$ —i.e., perform the substitution  $\psi[M \leftarrow S'(M), R \leftarrow S'(R)]$ . For instance, the term  $M(R(\text{ebp}) - 8)$  evaluates to the contents of memory at address  $R(\text{ebp}) - 8$ , i.e.,  $access(M, R(\text{ebp}) - 8)$ , which equals

$$access(update(M_0, R_0(\text{eax}), 5), R_0(\text{ebp}) - 8).$$

From the axiom for arrays, this simplifies to

$$ite(R_0(\text{eax}) = R_0(\text{ebp}) - 8, 5, M_0(R_0(\text{ebp}) - 8)).$$

Thus, the evaluation of  $\psi$  under  $S'$  yields

$$\left( \begin{array}{l} ite(R_0(\text{eax}) = R_0(\text{ebp}) - 8, \\ 5, M_0(R_0(\text{ebp}) - 8)) \\ + ite(R_0(\text{eax}) = R_0(\text{ebp}) - 12, \\ 5, M_0(R_0(\text{ebp}) - 12)) \end{array} \right) = 10 \quad (2)$$

This formula equals  $WLP(I, \psi)$  [30].

The process described above illustrates a general property: for any instruction  $I$  and formula  $\psi$ ,

$$WLP(I, \psi) = \psi[M \leftarrow S'(M), R \leftarrow S'(R)],$$

where  $S' = SE[I]S_{id}$  and  $SE[\cdot]$  denotes symbolic execution [30].

The next steps are to identify  $\alpha$  and to create a simplified formula  $\psi'$  that weakens  $WLP(I, \psi)$ . These are carried out simultaneously during a traversal of  $WLP(I, \psi)$ . We illustrate this on the example discussed above. Because the *ite*-terms in Eqn. (2) were generated from array accesses, *ite*-conditions represent the desired aliasing conditions. We traverse Eqn. (2), and for each term of the form  $ite(\varphi, t_1, t_2)$ , if  $\varphi$  holds in symbolic state  $S$ , then it is conjoined to  $\alpha$ , and the term is simplified to  $t_1$ . Otherwise, if  $\neg\varphi$  holds in  $S$ , then  $\neg\varphi$  is conjoined to  $\alpha$  and the term is simplified to  $t_2$ . If neither case holds, then the *ite* term and  $\alpha$  are left untouched.

In our example,  $R_0(\text{eax})$  equals  $R_0(\text{ebp}) - 8$  in symbolic state  $S$ ; hence, applying the process described above to Eqn. (2) yields

$$\begin{aligned} \psi' &= (5 + M_0(R_0(\text{ebp}) - 12) = 10) \\ \alpha &= \left( \begin{array}{l} (R_0(\text{eax}) = R_0(\text{ebp}) - 8) \\ \wedge (R_0(\text{eax}) \neq R_0(\text{ebp}) - 12) \end{array} \right) \end{aligned} \quad (3)$$

The formula  $\neg\alpha \vee \psi'$  (i.e.,  $\alpha \Rightarrow \psi'$ ) is the desired refinement predicate  $WLP_\alpha(I, \psi)$ .

This approach is *language-independent* because it isolates the consideration of the semantics of the instruction set to the computation of  $S' = SE[I]S_{id}$  in  $WLP(I, \psi)$ . All remaining steps are performed solely on formulas.<sup>2</sup>

It is true that the algorithm described above computes  $WLP(I, \psi)$  explicitly. However, this step alone does not cause an explosion in formula size—explosion is a consequence of repeated application of WLP. In our approach, the formula obtained via  $WLP(I, \psi)$  is immediately simplified to create first  $\psi' = \alpha \wedge WLP(I, \psi)$  and then  $\alpha \Rightarrow \psi'$ .

<sup>1</sup> In x86, `ebp` is the frame pointer, so if program variable `x` is at offset `-8` and `y` is at offset `-12`, this corresponds to the example discussed in §2, with `eax` playing the role of variable `p`.

<sup>2</sup> DASH and MCDASH need the symbolic-execution primitive  $SE[I]$  anyway for other steps of state-space exploration. Moreover, an implementation of  $SE[I]$  can be generated from a description of the semantics of an instruction set [30]; consequently, an implementation of  $WLP_\alpha(I, \psi)$  can be generated as well.

### 3.1.2 Byte-Addressable Memory

In the above discussion, we assumed that the memory map has type  $INT \rightarrow INT$ . In x86 machine code, memory is byte-addressable, so the actual type of the memory map is  $INT32 \rightarrow INT8$ . This complicates matters because accessing (updating) a 32-bit quantity in memory under the little-endian storage convention translates into four contiguous 8-bit accesses (updates); for instance, a 32-bit access can be expressed as follows:

$$\begin{aligned} \text{access\_32\_8\_LE\_32}(m, a) = & \\ \text{let } v1 = \text{Int8To32ZE}(m(a)) & \\ v2 = \text{Int8To32ZE}(m(a+1)) \ll 8 & \\ v3 = \text{Int8To32ZE}(m(a+2)) \ll 16 & \\ v4 = \text{Int8To32ZE}(m(a+3)) \ll 24 & \\ \text{in } (v4 \mid v3 \mid v2 \mid v1) & \end{aligned} \quad (4)$$

where  $\text{Int8To32ZE}$  converts an  $INT8$  to an  $INT32$  by padding the high-order bits with zeros, and “ $\mid$ ” denotes bitwise-or.

Let  $\text{update\_32\_8\_LE\_32}$  denote the similar operation for updating a map of type  $INT32 \rightarrow INT8$  under the little-endian storage convention. Note that when  $1 \leq |k_1 -_{INT32} k_2| \leq 3$ , we no longer have the property

$$\begin{aligned} \text{access\_32\_8\_LE\_32}(\text{update\_32\_8\_LE\_32}(M, k_1, d), k_2) \\ = \text{access\_32\_8\_LE\_32}(M, k_2). \end{aligned}$$

and hence it is invalid to simplify formulas by the rule

$$\begin{aligned} \text{access\_32\_8\_LE\_32}(\text{update\_32\_8\_LE\_32}(M, k_1, d), k_2) \\ \Rightarrow \text{ite}(k_1 = k_2, d, \text{access\_32\_8\_LE\_32}(M, k_2)). \end{aligned}$$

However, the four single-byte accesses on  $m$  in Eqn. (4) (i.e.,  $m(a)$ ,  $m(a+1)$ ,  $m(a+2)$ , and  $m(a+3)$ ), are  $\text{access}$  operations for which it is valid to apply Eqn. (1).

Returning to the example discussed in §3.1.1, where  $R_0(\text{eax})$  equals  $R_0(\text{ebp}) - 8$  in symbolic state  $S$ , we perform the same steps as before. First, the symbolic execution of  $I = \text{mov}[\text{eax}]$ , 5 starting from the identity symbolic state  $S_{id} = [M \mapsto M_0, R \mapsto R_0]$  results in the symbolic state

$$S' = [M \mapsto \text{update\_32\_8\_LE\_32}(M_0, R_0(\text{eax}), 5), R \mapsto R_0].$$

The formula  $\psi$  is now written as follows:

$$\begin{aligned} \text{access\_32\_8\_LE\_32}(M, R(\text{ebp}) - 8) \\ + \text{access\_32\_8\_LE\_32}(M, R(\text{ebp}) - 12) \\ = 10. \end{aligned}$$

To obtain  $\text{WLP}(I, \psi)$ , we evaluate  $\psi$  under  $S'$ , which yields the formula shown in Fig. 2—where for brevity we have introduced the notational shorthands  $p = R_0(\text{eax})$ ,  $x = R_0(\text{ebp}) - 8$ ,  $y = R_0(\text{ebp}) - 12$ ,  $*x = M_0(R_0(\text{ebp}) - 8)$ ,  $*y = M_0(R_0(\text{ebp}) - 12)$ , etc. The formula shown in Fig. 2 is the analog of Eqn. (2).

The step that uses symbolic state  $S$  to identify  $\alpha$  and create a simplified formula  $\psi'$  that weakens  $\text{WLP}(I, \psi)$  is now applied to the formula shown in Fig. 2, and produces

$$\psi' \stackrel{\text{def}}{=} 5 + \left( \begin{array}{l} 2^{24} * \text{Int8To32ZE}(*y + 3) \\ | 2^{16} * \text{Int8To32ZE}(*y + 2) \\ | 2^8 * \text{Int8To32ZE}(*y + 1) \\ | \text{Int8To32ZE}(*y) \end{array} \right) = 10,$$

and  $\alpha$  is the conjunction of the disequalities collected from the formula shown in Fig. 2:

$$\alpha \stackrel{\text{def}}{=} x + 3 \neq p + 3 \wedge \dots \wedge x + 3 \neq p \wedge \dots \wedge x \neq p + 3 \wedge \dots \wedge x \neq p \wedge y + 3 \neq p + 3 \wedge \dots \wedge y + 3 \neq p \wedge \dots \wedge y \neq p + 3 \wedge \dots \wedge y \neq p.$$

These are the analogs of Eqn. (3).

As before, the formula  $\neg\alpha \vee \psi'$  (i.e.,  $\alpha \Rightarrow \psi'$ ) is the desired refinement predicate  $\text{WLP}_\alpha(I, \psi)$ .

### 3.2 Local Variables

In this section, we describe an optimization necessary to improve the scalability of MCDASH-ICFG. Consider the program shown in Fig. 3. When DASH is executed on the source code to test reachability of the label `ERR`, it will perform refinement in its first two iterations. Next, it recursively calls itself on procedure `bar` with the target predicate  $\phi_{\text{src}}^1 = (g + l_1 = 5)$ . Inside `bar`, the first iteration again performs refinement to build the formula  $\phi_{\text{src}}^2 = (g + l_1 + l_2 = 5)$ .

The story changes when dealing with machine code. When we run MCDASH-ICFG on procedure `foo`, it does some refinements to build the formula  $\phi_{\text{mc}}^1 = (M(c_g) + M(\text{ebp} - 4) = 5)$  at the corresponding point to  $\phi_{\text{src}}^1$  (which is just after the call to `bar`). Here,  $c_g$  is the (constant) global address of the variable  $g$ .

Next, MCDASH-ICFG, like DASH, recursively calls itself on `bar`. Ignoring the return instruction, the first iteration would have the instruction “`pop ebp`” as the frontier, and would need to refine. The semantics of this instruction are that it assigns `ebp` the value  $M(\text{esp})$  and then increments `esp` by 4. Performing  $\text{WLP}_\alpha$  on  $\phi_{\text{mc}}^1$  results in  $(M(c_g) + M(M(\text{esp}) - 4) = 5)$ . The formula created at the point corresponding to  $\phi_{\text{src}}^2$  would be  $(M(c_g) + M(\text{ebp} - 4) + M(M(\text{ebp}) - 4) = 5)$ .

Note that the formula generated by MCDASH has a double memory dereference, even though the source code contains only ordinary accesses on variables. The reason for this is that MCDASH does not know that `ebp` is a callee-saved register in this program: at the `pop` instruction, it does not know that the value of `ebp` is restored to its value before the call to `bar`.

To reduce the complexity of the formulas that arise, we extend the notion of “ $\alpha$ ” to include the values of `esp` and `ebp`. If the frontier is  $(n, I, m)$  with  $S_n$  as the symbolic state, then in the computation of  $\text{WLP}_\alpha$  we check  $S_n$  to see if the values of `esp` and `ebp` are concrete (they may be symbolic). If they are concrete—say  $c_s$  and  $c_b$ —then we conjoin the constraint  $(R(\text{esp}) = c_s \wedge R(\text{ebp}) = c_b)$  to  $\alpha$ , and replace these registers with their constant values in the refinement predicate. Thus,  $\phi_{\text{mc}}^1$  would become

$$(R(\text{ebp}) = (c_{\text{stk}} - 4)) \Rightarrow (M(c_g) + M(c_{\text{stk}} - 8) = 5),$$

where  $c_{\text{stk}}$  is the starting value of `esp`. The result of  $\text{WLP}_\alpha$  on this formula across the `pop` instruction is:

$$\begin{aligned} (R(\text{esp}) = c_{\text{stk}} - 60) \wedge (M(c_{\text{stk}} - 60) = c_{\text{stk}} - 4) \Rightarrow \\ (g + M(c_{\text{stk}} - 8) = 5) \end{aligned}$$

The refinement predicate at the point corresponding to  $\phi_{\text{src}}^2$  is:

$$\begin{aligned} (R(\text{ebp}) = c_{\text{stk}} - 60) \wedge (M(c_{\text{stk}} - 60) = c_{\text{stk}} - 4) \Rightarrow \\ (M(c_g) + M(c_{\text{stk}} - 64) + M(c_{\text{stk}} - 8) = 5) \end{aligned}$$

By this means, the double memory dereference goes away, and the refinement predicate looks more like  $\phi_{\text{src}}^2$ , except that it has an extra constraint on the program stack.

The intuition behind using the concrete values of `esp` and `ebp` is similar to the intuition behind using aliasing condition  $\alpha$  in  $\text{WLP}_\alpha$ : the program is not expected to generate too many different aliasing conditions at a given program point, and its use greatly simplifies the refinement predicates. Similarly, at a particular program point in a given calling context, `esp` and `ebp` should not take on too many different values—in particular, in well-behaved programs they should each take on only a single value.

## 4. MCDASH-ICFG

In some cases, especially for stripped binaries, it is not possible to build an accurate description of the CFG of the program, without a full reasoning of the program’s semantics. Difficulties such as indirect jumps and indirect calls also show up in high-level languages.

$$\begin{aligned}
& \left( \begin{array}{l} 2^{24} * \text{Int8To32ZE}(\text{ite}(x + 3 = p + 3, 0, \text{ite}(x + 3 = p + 2, 0, \text{ite}(x + 3 = p + 1, 0, \text{ite}(x + 3 = p, 5, *(x + 3)))))) \\ 2^{16} * \text{Int8To32ZE}(\text{ite}(x + 2 = p + 3, 0, \text{ite}(x + 2 = p + 2, 0, \text{ite}(x + 2 = p + 1, 0, \text{ite}(x + 2 = p, 5, *(x + 2)))))) \\ 2^8 * \text{Int8To32ZE}(\text{ite}(x + 1 = p + 3, 0, \text{ite}(x + 1 = p + 2, 0, \text{ite}(x + 1 = p + 1, 0, \text{ite}(x + 1 = p, 5, *(x + 1)))))) \\ \text{Int8To32ZE}(\text{ite}(x = p + 3, 0, \text{ite}(x = p + 2, 0, \text{ite}(x = p + 1, 0, \text{ite}(x = p, 5, *x)))))) \end{array} \right) \\
+ & \left( \begin{array}{l} 2^{24} * \text{Int8To32ZE}(\text{ite}(y + 3 = p + 3, 0, \text{ite}(y + 3 = p + 2, 0, \text{ite}(y + 3 = p + 1, 0, \text{ite}(y + 3 = p, 5, *(y + 3)))))) \\ 2^{16} * \text{Int8To32ZE}(\text{ite}(y + 2 = p + 3, 0, \text{ite}(y + 2 = p + 2, 0, \text{ite}(y + 2 = p + 1, 0, \text{ite}(y + 2 = p, 5, *(y + 2)))))) \\ 2^8 * \text{Int8To32ZE}(\text{ite}(y + 1 = p + 3, 0, \text{ite}(y + 1 = p + 2, 0, \text{ite}(y + 1 = p + 1, 0, \text{ite}(y + 1 = p, 5, *(y + 1)))))) \\ \text{Int8To32ZE}(\text{ite}(y = p + 3, 0, \text{ite}(y = p + 2, 0, \text{ite}(y = p + 1, 0, \text{ite}(y = p, 5, *y)))))) \end{array} \right) \\
& = 10
\end{aligned}$$

**Figure 2.** The formula for  $\text{WLP}(I, \psi)$ , where  $\psi$  is  $\text{update\_32\_8\_LE\_32}(M, R(\text{ebp}) - 8) + \text{update\_32\_8\_LE\_32}(M, R(\text{ebp}) - 12) = 10$ , obtained by evaluating  $\psi$  on the symbolic state  $S' = [M \mapsto \text{update\_32\_8\_LE\_32}(M_0, R_0(\text{eax}), 5), R \mapsto R_0]$ . For brevity, the following notational shorthands are used in the formula:  $p = R_0(\text{eax})$ ,  $x = R_0(\text{ebp}) - 8$ ,  $y = R_0(\text{ebp}) - 12$ ,  $*x = M_0(R_0(\text{ebp}) - 8)$ ,  $*y = M_0(R_0(\text{ebp}) - 12)$ , etc.

```

procedure foo
. push ebp           ; save frame ptr on stack
. mov ebp, esp      ; ebp = esp
. sub esp, 56       ; make space for locals
. mov [ebp-4], 10   ; v1 = 10
. call bar          ; bar ( )
. mov eax, g        ; eax = g
. add eax, [ebp-4]  ; eax += v1
int g = 0;
void foo( ) {
. mov g, eax        ; g = eax
. cmp g, 5          ; g == 5?
. jnz short loc_5D ; jump if g != 5
. ERR: nop          ; skip
. loc_5D:
. mov esp, ebp     ; restore stack ptr
. pop ebp          ; restore frame ptr
. retn             ; return to callee
}

void bar( ) {
. push ebp           ; save frame ptr on stack
. mov ebp, esp      ; ebp = esp
. sub esp, 56       ; make space for locals
. mov [ebp-4], 20   ; v2 = 20
. mov eax, g        ; eax = g
. add eax, [ebp-4]  ; eax += v2
. mov g, eax        ; g = eax
. mov esp, ebp     ; restore stack ptr
. pop ebp          ; restore frame ptr
. retn             ; return to callee
}

```

**Figure 3.** An example program and its compiled x86 binary.

However, more acute is the problem of identifying procedures, because a binary need not follow any standard calling convention. For example, a program can use a return instruction to simulate a procedure call, and vice versa. For this reason, MCDASH-ICFG does not use a front-end for building a CFG.

One standard approach in the model-checking literature is to treat the program counter (PC) as data and use the CFG shown in Fig. 4(a), where  $\text{op}_{\text{any}}$  denotes any possible instruction (it stands for an abstraction of all possible instructions). However, using this approach with DASH has two difficulties: (i) A path in the abstract graph only conveys information about the number of instructions executed, not what those instructions are. Thus, during symbolic execution, if the PC value becomes a symbolic expression, then one would need to symbolically execute *all* of the possible instructions at PCs represented by the symbolic expressions. This can easily overwhelm the tool. (ii) The entire program would be treated as a single procedure. The interprocedural aspect of DASH is important for its scalability. We show how to solve each of these problems, in turn. For the first one, we show how one can make use of a technique from concolic execution [34, 21].

#### 4.1 Stealing Concrete Values

In concolic techniques for state-space exploration, the symbolic execution of a path  $\tau$  can steal values from a concrete execution of  $\tau$  to simplify the symbolic state. This has previously been used as a heuristic in tools for boosting test coverage. We show how to adapt the technique to work in a verification context. We explain the concept in source-level terms.

DASH uses symbolic execution to learn an under-approximation of the program’s behavior. We observe that one can relax the requirements of symbolic execution. Consider step 11 of Alg. 1, and suppose that  $\tau$  is the path (sequence of instructions) that the concrete execution took to reach state  $c$ . Let  $\text{SE}[\tau]S_{id}$  be the symbolic state obtained after symbolically executing the path  $\tau$  from the initial identity symbolic state. The DASH algorithm remains correct, while ensuring progress, if the following two properties are satisfied: (i)  $S = \text{SE}[\tau]S_{id}$  generalizes  $c$ , i.e., there is some assignment to the input symbolic constants for which  $S$  equals  $c$ , and (ii) if  $S' = \text{SE}[\tau; I]S_{id}$  is feasible then the path  $\tau; I$  must be executable. The first property ensures that if refinement is performed, then the frontier changes in the next iteration. (In particular, it ensures that  $c$  is not a witness for  $n'$  in Fig. 1(c).) The second property ensures that if  $S'$  is feasible, and we run a test using the input obtained from  $S'$ , then the test must follow  $(\tau; I)$ —and thus make progress towards the target. Next, we show how stealing concrete values still preserves these two properties.

*Example.* Suppose that  $I' = (z = x \ll y)$  has to be symbolically executed, where  $\ll$  is the bitwise left-shift operator. If the current symbolic state has path constraint  $\varphi$  and symbolic map  $S_{\text{before}} = [x \mapsto f(\text{inp}), y \mapsto g(\text{inp})]$ , for some functions  $f$  and  $g$  over the input symbolic constants, then the result of executing  $I$  updates the value of  $z$  to  $(f(\text{inp}) \ll g(\text{inp}))$ .

In concolic execution, one avoids creating complex symbolic expressions (because the theorem prover has to later check for satisfiability of formulas over these expressions) by using concrete information. Suppose that we wish to avoid having a symbolic value for the shift-argument of  $\ll$ . Then, if the concrete state before the execution of  $I'$  is  $[y \mapsto 2, \dots]$ , concolic execution techniques would “steal” the value of  $y$  from the concrete state [34] and simplify the resulting symbolic map to:

$$S_{\text{after}} = [x \mapsto f(\text{inp}), y \mapsto g(\text{inp}), z \mapsto f(\text{inp}) \ll 2]$$

This technique does not satisfy the latter of the two properties mentioned above: if  $\text{inp}$  has just one symbolic constant  $s_0$ ,  $f = (s_0 + 1)$ ,  $g = s_0$ ,  $\varphi = \text{true}$ ,  $I'$  was the last instruction of  $\tau$ , and  $I$  is a branch that tests  $(z == 4)$ , then executing  $I$  on the symbolic state  $S_{\text{con}}$  results in the path constraint  $(s_0 + 1) \ll 2 == 4$ , which is satisfiable with  $s_0 == 0$ . Running a test with this input would result in the concrete state  $c_{\text{before}} = [x \mapsto 1, y \mapsto 0]$  before  $I'$

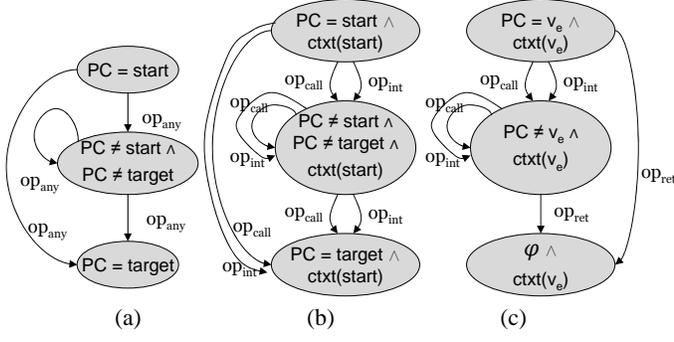


Figure 4. Abstract graphs created by MCDASH-ICFG

(obtained by substituting 0 for  $s_0$  in  $S_{\text{before}}$ ). After the execution of  $I'$ , the branch  $I$  cannot be taken.

For MCDASH, we change the process of stealing concrete values. Whenever the concrete value of a variable  $y$  is stolen, say it is  $c_y$ , then the symbolic map is updated as before, but the constraint  $y == c_y$  is symbolically evaluated and added to the path constraint. In the above example, the constraint  $y == 2$  evaluates to  $s_0 == 2$  under  $S_{\text{before}}$ , and is conjoined to  $\varphi$ . The reader can verify that the execution of  $I$  will result in an infeasible state. The basic idea behind our approach is to treat the process of stealing concrete values as if there was a fictitious branch condition  $y == c_y$  in the path. The concrete execution certainly satisfies this branch and proceeds through it, and the symbolic execution picks up the appropriate constraint, which also allows it to simplify its symbolic map.

For MCDASH-ICFG, the symbolic execution keeps stealing the value of the PC from the concrete state at every step. This ensures that the symbolic map always has a concrete value for the PC, hence it only has to symbolically execute a single fixed instruction at each step.

#### 4.2 Interprocedural without CFGs

The abstract graphs constructed by DASH at any stage only refer to nodes of a single procedure. This allows it to keep these graphs to a manageable size. In the absence of any information about procedures, MCDASH-ICFG would have to use a single graph to capture the abstraction of an entire program.

We avoid this problem by defining *procedural contexts* (CTXTs), which serve as a substitute for the notion for a procedure. There is one context  $\text{CTXT}(v)$  for each possible value  $v$  of the PC. The context  $\text{CTXT}(v)$  roughly serves as a procedure identifier for the one that begins at  $\text{PC} = v$ . The concrete state of the program is instrumented with a stack of CTXTs that is manipulated by the concrete execution. We use  $c.stk$  to refer to the stack associated with concrete state  $c$ .

If a test has to be started from state  $c$ , and  $\text{PC}(c) = v$ , then  $c.stk$  is initialized to  $[\text{CTXT}(v)]$ . If the execution of a call instruction takes state  $c_1$  to  $c_2$  then  $c_2.stk = \text{push}(\text{CTXT}(\text{PC}(c_2)), c_1.stk)$ . If the execution of a return instruction takes state  $c_3$  to  $c_4$  then  $c_4.stk = \text{pop}(c_3.stk)$ , provided  $c_3.stk$  has at least two elements. In all other cases, the stack is left unchanged. (For well-behaved programs, this stack identifies the current procedure along with its calling context.) Similar manipulation is performed for symbolic execution. We add an additional type of constraint in our logic:  $\text{ctx}(v)$ , which is satisfied by a concrete state  $c$  only when the top element of  $c.stk$  is  $\text{CTXT}(v)$ .

The initial abstract graph constructed by MCDASH-ICFG is shown in Fig. 4(b). In the graph,  $\text{op}_{\text{call}}$  is an abstraction of all call instructions,  $\text{op}_{\text{ret}}$  is an abstraction of all return instructions, and  $\text{op}_{\text{int}}$

is all other instructions. Thus, MCDASH-ICFG knows the CTXT-stack manipulations that each of the three kinds of instructions can perform.

During the execution of MCDASH-ICFG, suppose that the frontier has  $\text{op}_{\text{call}}$  as the instruction, the formula on its target is  $\varphi$ , and the concrete state at its source is  $c$ . The first step is to identify the procedure being called. If the next instruction executed by  $c$  is not a call instruction, then we refine using the predicate  $\text{PC} == v_c$ , where  $v_c$  is the PC value of  $c$ . This refinement will result in the removal of the  $\text{op}_{\text{call}}$  edge (because no call instruction can fire from a state that satisfies  $\text{PC} == v_c$ ). Otherwise, let  $v_e$  be the PC after the call instruction is executed at  $c$ . We steal the PC value  $v_e$  using the method discussed in the previous section. Next, MCDASH-ICFG calls itself recursively on the abstract graph shown in Fig. 4(c) to see if  $\varphi$  is reachable or not. In essence, we use call and return instructions to figure out contexts that MCDASH should verify in isolation.

### 5. MCDASH-SMC

For self-modifying code (SMC), the association of a PC value with the instruction at that PC is no longer fixed. We extended MCDASH to incorporate the decoding relationship between a sequence of bytes in memory and the instruction that those bytes represent. To do this two strategies were possible; however, with the present MCDASH implementation we were only able to try the first:

1. Similar to MCDASH-ICFG, we simplify the cases when the PC value or the bytes in memory at that PC are symbolic expressions. During symbolic execution, we steal the PC value as well as the memory bytes at that PC from the concrete state. This preserves soundness, and ensures that symbolic execution only has to execute a fixed instruction at each step (i.e., if MCDASH-SMC returns a proof that a property holds, then it indeed holds).
2. The alternative is to only steal the PC values, but allow the instruction to be symbolic. To accomplish this, the decoding relationship (byte-sequence to instruction), as well as the instruction semantics would need to be expressed symbolically so that symbolic execution can form expressions and constraints over them. We leave this approach for future work.

Except for the above change, the MCDASH-SMC algorithm is identical to MCDASH-ICFG.

MCDASH-SMC can verify the compiled version of the C code shown in Fig. 5<sup>3</sup>. The variable `inp` is the input to the program. The array `code` stores the binary encoding of the instructions shown in comments above it. This code increments the value of register `ecx`. `main` calls this code, and after it returns, `code` is modified to change the immediate argument of the `add` instruction to `-1`. Thus, the next time `code` is executed, the value of `ecx` is decremented by 1.

MCDASH-SMC is able to verify that target label `ERR` is not reachable. In MCDASH-ICFG, predicates of the form  $(\text{PC} == v)$  help it learn control-flow information because nodes of the abstract graph that have such a constraint only have fixed successors (unless the instruction at that PC is `call`, `return` or an indirect jump). Similarly, in MCDASH-SMC, the predicates that help build control-flow information are of the form  $(\text{PC} == v) \wedge \text{decode}(v, I)$ , where the latter is a constraint that the contents of the memory at location  $v$  represent instruction  $I$ . For the example in Fig. 5, MCDASH-SMC is able to pick up the constraints  $(\text{PC} == c) \wedge \text{decode}(c, \text{"add ecx, 1"})$  and  $(\text{PC} == c) \wedge \text{decode}(c, \text{"add ecx, -1"})$ , allowing it to explore the possibility of executing different instructions at the same PC.

<sup>3</sup>This example is adapted from the one described in <http://www.acm.org/src/Joy/joy.htm>

```

void main(int inp) {
    int old = inp;
    _asm { mov ecx, inp }
/*
 * add ecx, 1;
 * ret;n;
 */
unsigned char code[] = {0x83,
    0xc1, 0x01, 0xc3};
    _asm { mov inp, ecx }
    if(inp != old) {
        ERR: return;
    }
}

```

Figure 5. Self-Modifying Code

## 6. Implementation

The MCDASH implementation has been structured so that it can be retargeted to different languages easily. The core components of the system are language-independent in two different dimensions:

1. The MCDASH driver implements Alg. 1. It is structured so that one only needs to provide an implementation of the concrete and symbolic execution of a language, and a few other primitives (e.g.,  $WLP_\alpha$ ). Consequently, this component of the system can be used for source-level languages or for machine-code languages.
2. For machine-code languages, we have used two tools that *generate* the required implementation of the concrete semantics and the symbolic-analysis primitives from descriptions of the syntax and semantics of an instruction set of interest.

The abstract syntax and concrete semantics of an instruction set are specified using a language called TSL (Transformer Specification Language) [31]. Decoding (i.e., translation of binary-encoded instructions to abstract syntax trees) is specified using a tool called ISAL (Instruction Set Architecture Language).<sup>4</sup> The relationship between ISAL and TSL is similar to the relationship between Flex and Bison. With Flex and Bison, a Flex-generated lexer passes tokens to a Bison-generated parser. In our case, the TSL-defined abstract syntax serves as the formalism for communicating values—namely, instructions’ abstract syntax trees—between the two tools.

Compared with other specification languages for instruction sets, TSL has one unique feature: from a *single* specification of the concrete semantics of an instruction set a *multiplicity* of static-analysis, dynamic-analysis, and symbolic-analysis components can be *generated automatically*. The TSL system consists of two parts:

- The TSL language for specifying an instruction set’s abstract syntax and concrete semantics. TSL is a strongly typed, first-order functional language with a datatype-definition mechanism for defining recursive datatypes, plus deconstruction by means of pattern matching.
- The TSL compiler, which translates a specification to a common intermediate representation (CIR). The CIR generated for a given TSL specification is a C++ template that can be used to create multiple analysis components by instantiating the template in different ways.

TSL has two classes of users: (1) instruction-set specifiers, and (2) analysis developers. The former use the TSL language to specify the concrete semantics of different instruction sets; the latter create new analyses by instantiating the CIR in different ways.

**Specifying an Instruction Set.** Much of what an instruction-set specifier writes in a TSL specification is similar to writing an interpreter for an instruction set in first-order ML [24]. One specifies (i) the abstract syntax of the instruction set, by defining the con-

<sup>4</sup> ISAL also handles other kinds of concrete syntactic issues, including (a) *encoding* (abstract syntax trees to binary-encoded instructions), (b) *parsing assembly* (assembly code to abstract syntax trees), and (c) *assembly pretty-printing* (abstract syntax trees to assembly code).

structors for a (reserved, but user-defined) type *instruction*; (ii) a type for concrete states, by defining—e.g., for 32-bit Intel x86—the type *state* as a triple of maps:

$$state : State(INT32 \rightarrow INT8, reg32 \rightarrow INT32, flag \rightarrow BOOL);$$

where *INT32* and *INT8* refer to 32-bit and 8-bit integers, respectively, and *reg32* and *flag* refer to a type for the names of 32-bit registers and a type for the names of condition-codes, respectively; and (iii) the concrete semantics of each instruction by writing a TSL function

$$state\ interInstr(instruction\ I, state\ S) \{ \dots \};$$

**Semantic Reinterpretation.** Each analysis is defined by reinterpreting the constructs of the TSL meta-language. TSL’s meta-language supports a fixed set of base-types; a fixed set of arithmetic, bitwise, relational, and logical operators; and a facility for defining map-types. An analysis developer defines a new analysis component by (i) redefining (in C++) the TSL base-types (*INT32*, *INT8*, *BOOL*, etc.), and (ii) redefining (in C++) the primitive operations on base-types ( $+_{INT32}$ ,  $+_{INT8}$ , etc.). These are used to instantiate the CIR template. This implicitly defines an alternative interpretation of each expression and function in an instruction-set’s concrete semantics (including *interInstr*), and thereby yields an alternative semantics for an instruction set from its concrete semantics.

For MCDASH, TSL is used to create several useful reinterpretations of an instruction set:

- By instantiating the CIR with a reinterpretation that performs the standard interpretation (in C++) of the TSL operators, we obtain the instruction interpreter for concrete execution.
- By instantiating the CIR with a reinterpretation that instantiates *INT32*, *INT16*, and *INT8* as the types of symbolic expressions that denote 32-bit, 16-bit, and 8-bit values, respectively, in the input language of an SMT solver, and operations (such as  $+$ ,  $*$ ,  $==$ , etc.) as simplifying constructors<sup>5</sup> we obtain a semantics suitable for symbolic execution. (In our implementation, we used the Yices input language [18].)
- A third reinterpretation creates a primitive for performing *WLP* [30]. (As explained in §3.1, *WLP* is used a subroutine in the implementation of  $WLP_\alpha$ .)

These reinterpretations are used as subroutines in MCDASH’s components for concrete execution, symbolic execution, and *WLP* computation.

In MCDASH-ICFG, decoding of instructions is done all at once, at ICFG-construction time. In MCDASH-ICFG and MCDASH-SMC, decoding of instructions is performed instruction-by-instruction, as concrete or symbolic execution proceeds.

## 7. Experiments

We designed our experiments to test how competitive MCDASH is against source-level tools. We compared against DASH on examples from [23]<sup>6</sup> on which an earlier version of DASH was tested. These are hand-crafted examples designed to illustrate various aspects of the DASH algorithm. The later version of DASH [6] was tested on device drivers. We could not use these examples because we did not have the harnesses and the OS stubs for the drivers.

The examples are all written in C. We compiled them and ran MCDASH-ICFG and MCDASH-ICFG on the resulting object file (without using the symbol-table information). The source code does not use pointers, but the compiled binary manipulates addresses to access local variables from the stack. The results are

<sup>5</sup> Straightforward simplifications are performed; e.g.,  $a == a$  simplifies to true, etc.

<sup>6</sup> These are available from that paper’s author’s homepage <http://www.cse.iitb.ac.in/~bhargav/synergy>.

shown in Fig. 6. Comparing with the timing numbers in [23], MCDASH is in the same range, except for a couple of examples. Moreover, surprisingly, MCDASH-ICFG was sometimes faster than MCDASH-ICFG. This was because the absence of a CFG forced its search to proceed in a different manner than MCDASH-ICFG. And, as a result, it got lucky in finding the desired loop invariants faster.

## 8. Related Work

### Machine-Code Analyzers Targeted at Finding Vulnerabilities.

A substantial amount of work has been carried out on analysis techniques to detect security vulnerabilities by analyzing source code for a variety of languages [36, 11, 32, 37]. Less work has been done on vulnerability detection for machine code. Kruegel et al. [27] developed a system for automating mimicry attacks. Their tool uses symbolic execution of x86 machine code to discover attacks that can give up and regain execution control by modifying the contents of the data, heap, or stack so that the application is forced to return control to injected attack code at some point after a system call has been performed. Cova et al. [14] used this platform to apply symbolic execution to the problem of detecting security vulnerabilities in x86 executables.

Both Godefroid et al. [22] and Brumley et al. [7] have created tools for performing concolic execution on x86 machine code. Concolic execution combines concrete execution and symbolic execution with the goal of finding inputs that increase test coverage. Calls to an SMT solver are used to obtain inputs that force previously unexplored branch directions to be taken. In contrast, DASH and MCDASH combine concrete execution and symbolic execution with abstraction; they are goal-directed: they try to refute the claim that there is no path from program entry to a given goal state.

In addition, the implementations of the other machine-code analyzers cited above are x86-specific, whereas our work can be retargeted to a new instruction set merely by writing a TSL specification and applying the TSL compiler.

**Self-Modifying Code.** The work on MCDASH-SMC addresses a problem that has been almost entirely ignored by the PL research community. There is one paper on SMC by Gerth from 1991 [20], and one recent paper by Cai et al. [9]. However, both of those papers concern proof systems for reasoning about SMC.

In contrast, MCDASH-SMC can analyze SMC automatically. As far as we know, MCDASH-SMC is the first model checker to address verifying (or detecting flaws in) SMC. It is also possible to generate versions of MCDASH-SMC for different instruction sets from descriptions of an instruction set's syntax and semantics.

## References

- [1] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, P. Hawkins, and B. Hackett. An overview of the Saturn project. In *PASTE*, 2007.
- [2] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *CC*, pages 5–23, 2004.
- [3] G. Balakrishnan and T. Reps. DIVINE: DIScovering Variables IN Executables. In *VMCAI*, 2007.
- [4] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys*, 2006.
- [5] T. Ball and S. Rajamani. The SLAM toolkit. In *CAV*, 2001.
- [6] N. Beckman, A. Nori, S. Rajamani, and R. Simmons. Proofs from tests. In *ISSTA*, 2008.
- [7] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, P. Pooankam, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In W. Lee, C. Wang, and D. Dagon, editors, *Botnet Analysis and Defense*, pages 65–88. Springer, 2008.
- [8] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30:775–802, 2000.
- [9] H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. In *Prog. Lang. Design and Impl.*, 2007.
- [10] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE*, 2003.
- [11] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *CCS*, pages 235–244, Nov. 2002.
- [12] CodeSonar, GrammaTech, Inc., [www.grammatech.com/products/codesonar](http://www.grammatech.com/products/codesonar).
- [13] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE*, pages 439–448, 2000.
- [14] M. Cova, V. Felmetger, G. Banks, and G. Vigna. Static detection of vulnerabilities in x86 executables. In *ACSAC*, 2006.
- [15] Coverity Prevent. [www.coverity.com/products/prevent\\_analysis\\_engine.html](http://www.coverity.com/products/prevent_analysis_engine.html).
- [16] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI*, 2002.
- [17] S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *POPL*, pages 12–24, 1998.
- [18] B. Dutertre and L. de Moura. Yices: An SMT solver, 2006. <http://yices.csl.sri.com/>.
- [19] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, pages 1–16, 2000.
- [20] R. Gerth. Formal verification of self modifying code. In Y. Liu and X. Li, editors, *Proc. Int. Conf. for Young Computer Scientists*, pages 305–311, Beijing, China, 1991. Int. Acad. Pub.
- [21] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Prog. Lang. Design and Impl.*, 2005.
- [22] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.
- [23] B. Gulavani, T. Henzinger, Y. Kannan, A. Nori, and S. Rajamani. SYNERGY: A new algorithm for property checking. In *FSE*, 2006.
- [24] E. Harcourt, J. Mauney, and T. Cook. Functional specification and simulation of instruction set architectures. In *Proc. Int. Conf. on Sim. and Hardw. Desc. Langs.* SCS Press, 1994.
- [25] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *STTT*, 2(4), 2000.
- [26] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Princ. of Prog. Lang.*, pages 58–70, 2002.
- [27] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *USENIX Sec. Symp.*, 2005.
- [28] J. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *PLDI*, pages 291–300, 1995.
- [29] K. R. M. Leino. Efficient weakest preconditions. *Inf. Proc. Let.*, 93(6):281–288, 2005.
- [30] J. Lim, A. Lal, and T. Reps. Symbolic analysis via semantic reinterpretation. In *Spin Workshop*, 2009.
- [31] J. Lim and T. Reps. A system for generating static analyzers for machine instructions. In *CC*, 2008.
- [32] B. Livshits and M. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Sec. Symp.*, 2005.
- [33] R. Lo, K. Levitt, and R. Olsson. MCF: A malicious code filter. *Computers & Society*, 14(6):541–566, 1995.
- [34] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *FSE*, 2005.
- [35] M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. In *FSE*, 1999.
- [36] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, 2000.

Program			ICFG Available				ICFG Not Available			
Name	#Instrs.	Outcome	CE	SE	Ref	time	CE	SE	Ref	time
barber	196	proof	2	14	94	21.6	1	23	59	2.8
berkeley	75	proof	5	7	22	50.1	4	37	81	3.7
cars	108	proof	4	10	91	16.1	3	33	87	6.2
efm	133	test	10	49	325	128.8	7	81	371	72.0
fig6	17	proof	2	4	13	5.6	2	20	31	9.0
fig7	17	test	2	1	0	0.1	2	6	5	0.2
fig8	61	proof	2	3	13	0.6	1	4	5	0.7
fig9	16	proof	1	3	13	1.4	1	14	25	3.0
prog2	21	proof	1	3	17	1.0	1	23	38	3.2
prog3	18	proof	1	3	14	0.7	1	20	32	2.0
prog4	38	proof	2	6	30	5.8	2	36	63	21.0
prog5	22	proof	1	3	18	1.0	1	24	40	3.1
test1	23	proof	2	5	27	4.2	2	30	86	9.8
test2	32	proof	2	9	48	4.8	2	46	138	19.8

**Figure 6.** MCDASH experiments. The columns, in order, are: the number of instructions (#Instrs); whether MCDASH returned a proof or a counterexample (Outcome); the number of concrete executions (CE); the number of symbolic executions (SE), which also equals the number of calls to the theorem prover; the number of refinements (Ref), which also equals the number of  $WLP_\alpha$  computations; and the total time taken in seconds. The experiments were run on a Intel P4 3.2GHz machine with 3.3GB RAM.

[37] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Sec. Symp.*, 2006.