ARE: Automated Reverse Engineering of Machine Code

Vu Le, Quy Messiter, Robert Ross, and Gregory Sadosuk
BAE Systems
4301 N. Fairfax Drive, Suite 800, Arlington, VA, 22203
1-703-284-84{69, 85, 97, 64}
{vu.le, quy.messiter, robert.b.ross, gregory.sadosuk}@baesystems.com

ABSTRACT
Binary code analysis is a challenging problem, but it is essential for many applications such as program analysis, verification, security, and optimization. BAE Systems developed the Automated Reverse Engineering (ARE) system for binary analysis to detect potential flaws in software and find inputs that steer program execution to reach areas of interest such as unexplored instructions, dangerous operations, or desired functionality. At a high level, the system provides a unique combination of static and dynamic analyses, efficient program databases, flexible constraint systems, and advanced software defect detection. This paper describes the major components of the ARE system, explains how they work together, and illustrates the steps of a demonstration.

Categories and Subject Descriptors
F.3.2 **[Logics and Meanings of Programs]**: Semantics of Programming Languages – *algebraic approaches to semantics*, *program analysis*.

General Terms
Algorithms, security, system

Keywords
Assembly language; binary code; reverse engineering; intermediate language; cyber security; malware; pattern detection; symbolic execution

Acknowledgments
This material is based upon work supported by Defense Advanced Research Projects Agency (DARPA) and SPAWAR Systems Center Pacific (SSC Pacific) under Contract No. N66001-13-C-4047. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA) or SPAWAR Systems Center Pacific (SSC Pacific). Distribution Statement A. Approved for Public Release; Distribution Unlimited. Cleared for Open Publication on 4/10/14. Non-Technical Data – Releasable to Foreign Persons.

1. INTRODUCTION
Despite tremendous advances in verification technology, software tends to exhibit, or be capable of, undesirable behaviors. Even if the source code for a program is available, it may use intentionally compromised closed-source libraries or accidentally acquire unexpected behaviors during compilation. Most current technologies do not address these problems since they rely on source code or other documentation. ARE is a system for static and dynamic analysis of the machine language for applications. Static analysis is performed by inspecting the binary files from the target application and translating native instructions into an intermediate language named PREIL. The PREIL instructions are then transformed and aggregated by TMF to become algebraic expressions suitable for forming searchable patterns of software behaviors. Dynamic analysis is performed by monitoring the target at runtime to record executed instructions and changes to the machine's state (e.g., memories and registers). Next, analytics extract data flows and locate bugs. The target application is then rerun, with inputs provided by a constraint solver, to approach areas of interest for verification and greater code coverage.

The primary objective of ARE, to analyze machine language, is shared by other systems such as BAP [1] and TSL [2], but ARE offers benefits such as (1) reconciliation of static preprocessing with dynamic traces to get fewer false negatives, (2) a simpler intermediate representation for instructions to support faster analysis tools, (3) algebraic expressions for semantic relationships to improve scalability, (4) deep inspection via full system simulators to improve completeness, (5) efficient management of large yet transient datasets via NoSQL databases, (6) static bug search to limit how many traces need to be checked, and (7) constraint queries such as finding maximal input values.

ARE was briefly explained in [3]. The following sections further describe ARE's system architecture, its main components, and how they operate together. We conclude with a demonstration of a simple test case to illustrate the main characteristics of ARE.

2. SYSTEM ARCHITECTURE
ARE automates the process of reverse engineering binary code for various architectures. Its heterogeneous components are split into five groups: static analysis, dynamic analysis, databases, bug search, and constraint system. Information gathered from static and dynamic analyses is stored in a NoSQL database and queried by bug search tools to find vulnerabilities, and by a constraint system, to provide inputs that lead execution to desirable paths.

Static analysis consists of (1) Hex-Ray's IDA Pro, to analyze binaries statically, (2) a preprocessor, to translate native instructions into platform-independent PREIL instructions, (3) IDA plugins, to extract instructions, basic blocks, functions, modules, and Control Flow Graphs (CFGs), and (4) TMF, to form higher-level semantic expressions from PREIL lists. Dynamic analysis consists of (1) system emulators, such as Simics and QEMU, which produce traces during executions and (2) Trace Parser, a tool that processes traces, e.g., to build Information Pedigree Graphs (IPGs) for dynamic taint queries. The static and dynamic information is stored in a distributed database which is capable of handling very large datasets. The database is queried by bug search tools to detect vulnerabilities such as buffer overflows, double frees, use after frees, and uninitialized memory reads. When potential bugs are detected the constraint system solves path constraints to determine input values that lead execution to those areas. The following sections further describe the components of the ARE system shown in Figure 1.

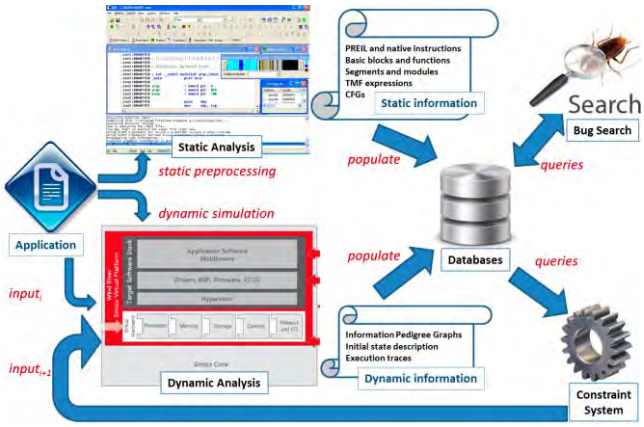

Figure 1. ARE Architecture

3. STATIC ANALYSIS

Static analysis components pertain to entire binaries. The information they provide applies to all possible execution paths.

3.1 Preprocessor

The ARE preprocessor analyzes binary code and emits platform-independent structures that are used by other ARE components. Basic analysis is handled by IDA Pro, and advanced analysis is handled by custom software and IDA plugins. When feasible, the preprocessor resolves external references to maximize coverage of the application space. When available, the preprocessor uses information gleaned from traces to rebase binaries (so instruction addresses match), to further resolve external references (so obscured indirect addresses are captured), and to reconcile differences (such as instructions that IDA Pro erroneously marked as data). The preprocessor translates each native instruction into PREIL so other components only need to consider a simple, platform-independent representation. Also, the preprocessor produces platform-independent information about binary modules, functions, and basic blocks. Preprocessor results are typically computed once and reused many times.

3.2 PREIL

PREIL is a simple intermediate representation with only 20 instructions. Each PREIL instruction has 3 operands and no side effects. PREIL is an enhancement to REIL, Google's Reverse Engineering Intermediate Language [4]. PREIL offers improved support for static analysis and instructions that REIL does not accurately translate. Since it is platform independent, ARE can target diverse platforms such as x86 and MIPS.

An example of each PREIL instruction is provided in Table 1. Each nonempty PREIL operand begins with either an "r" (for register) or "i" (for integer literal) followed by a size in bits, e.g., "r32" for a 32-bit register. PREIL registers are either persistent registers (denoted r_i) or temporary registers (denoted t_i) whose scopes are restricted to the PREIL list for a native instruction.

Since PREIL does not have side effects, the instruction translation process exposes implicit operations in native languages such as x86 assembly. For instance "push *eax*" involves pushing register *eax* to the stack after decrementing the stack pointer *esp* by 4 bytes. The decrement operation is implied by the instruction but explicitly presented in PREIL, as seen in the PREIL list below.

sub[r32 *esp*][i32 4][r32 *esp*] *(1)*
stm[r32 *eax*][][r32 *esp*] *(2)*

Table 1. Example PREIL instructions

PREIL Instruction	Meaning (C Interpretation)
add[r32 4][i32 r_1][r64 t_1]	Addition: $t_1 = 4 + r_1$;
and[r32 t_1][i32 F][r32 t_2]	Bitwise AND: $t_2 = t_1$ & 0xF;
bisz[r32 t_1][][r8 t_2]	Boolean is-zero: $t_2 = t_1$? 0 : 1;
div[r32 t_1][r32 t_2][r32 t_3]	Unsigned division: $t_3 = t_1 / t_2$;
ifm(r8 r_1)[r8 t_1][i8 9][r32 t_2]	Conditional store to memory: $*t_2 = r_1$? t_1 : 9;
ifr(r1 t_1)[r8 r_1][r8 r_2][r8 r_3]	Conditional store to register: $r_3 = t_1$? r_1 : r_2;
jcc[r32 t_0][][r32 t_1]	Jump conditional: if (t_0) goto t_1;
ldm[r32 t_1][][r8 t_2]	Load from memory: $t_2 = *t_1$;
lsh[r32 t_1][i16 4][r64 t_2]	Left shift: $t_2 = t_1 << 4$;
mod[r32 t_0][r32 t_1][r32 t_2]	Modulo: $t_2 = t_0$ % t_1;
mul[r32 t_1][r32 t_2][r64 t_3]	Unsigned multiply: $t_3 = t_1 * t_2$;
nop[][][]	No operation: ;
or[r32 t_1][i32 4][r32 t_2]	Bitwise OR: $t_2 = t_1 \mid 4$;
rsh[r32 t_1][i32 5][r64 t_2]	Logical right shift: $t_2 = t_1 >> 5$;
stm[r8 t_1][][r32 t_2]	Store to memory: $*t_2 = t_1$;
str[r32 t_1][][r32 t_2]	Store to register: $t_2 = t_1$;
sub[r32 t_1][i32 4][r64 t_2]	Subtract: $t_2 = t_1 - 4$;
undef[][][r32 r_1]	Undefine a register: uint32_t *u*; $r_1 = u$;
unkn[][][]	Unknown operation: void *f* (); *f* ();
xor[r32 t_1][i32 4][r32 t_2]	Bitwise XOR: $t_2 = t_1$ ^ 4;

3.3 TMF

TMF transforms abstract binary code into algebraic expressions without losing any relevant semantic information. TMF first models each PREIL instruction as an algebraic expression and then aggregates them to form TMF expressions. Aggregation occurs at many levels, e.g., instruction, basic block, and function.

Modeling a single PREIL instruction captures the semantics of the instruction algebraically. The transformation varies accordingly to the PREIL opcode. Every opcode except jcc, nop, undef, and unkn has a lossless transformation. Table 2 shows some examples.

Table 2. Example TMF transformations

PREIL Instruction	TMF Expression
and[r32 t_1][i32 F][r32 t_2]	$t_2 = t_1$ & 0xf
ifm(r8 r_1)[r8 t_1][i8 9][r32 t_2]	$[t_2] = r_1$? t_1 : 9
ifr(r1 t_1)[r8 r_1][r8 r_2][r8 r_3]	$r_3 = t_1$? r_1 : r_2
ldm[r32 t_1][][r8 t_2]	$t_2 = [t_1]$
stm[r8 t_1][][r32 t_2]	$[t_2] = t_1$

Brackets *[* and *]* indicate pointer dereferencing so *[esp]* is the value at the memory address in stack pointer *esp*. For example, consider the TMF transforms of PREIL instructions *(1)* and *(2)*:

sub[r32 *esp*][i32 4][r32 *esp*] transforms to *esp* = *esp* - 4 *(3)*
stm[r32 *eax*][][r32 *esp*] transforms to *[esp]* = *eax* *(4)*

Aggregation of the expressions in *(3)* and *(4)* produces the expression in *(5)* where the *esp* in *(4)* is replaced with the *esp* - 4 in *(3)*:

esp = *esp* - 4 ; *[esp]* = *eax* transforms to *[esp* - 4*]* = *eax* *(5)*

Notice that the single expression in *(5)* describes two actions.

4. DYNAMIC ANALYSIS

Dynamic analysis components pertain to individual traces. Their scope is limited, but that allows inspections to be more thorough.

4.1 Simulators

ARE uses Wind River's Simics to simulate Unix-like systems, Intel's Pin to instrument Windows-based applications when full system simulation is not necessary, and QEMU to emulate MIPS systems. Though ARE uses different simulators for different platforms, they all fulfill the same goals such as determining the initial state and capturing state changes in fixed format traces.

Since simulators are time consuming and resource intensive, snapshots are used to reduce the work for related runs. Snapshots also increase consistency, e.g., since loaded libraries will be at the same addresses despite ASLR. Furthermore, snapshots allow executables to be setup such that jump tables are resolved, hot-patching is complete, packed software is unpacked, and so on.

For faster trace generation, ARE includes an Emulated Tracer that produces the same traces as a faithful simulator would when the PREIL instructions are complete and accurate. A unique feature of the Emulated Tracer is its ability to begin traces at arbitrary addresses, including those not yet reachable by known inputs.

4.2 Trace Parser

A trace consists of a sequence of records. Each record pertains to a native instruction and contains (1) a sequence number to help orient program slices, (2) a thread identifier to distinguish different uses of the same register names, (3) the instruction address to associate with the static representation, (4) the machine language bytes to ensure that the static representation remains current, (5) an optional assembly language description to help analysts, and (6) lists of memory and register reads and writes where the stated values for reads (resp. writes) are before (resp. after) instruction execution. The Trace Parser processes traces for analyses such as CSI (Constraint Solver Interface), IPG (Information Pedigree Graph), and DBS (Dynamic Bug Search).

CSI describes an execution path, i.e., a sequence of instructions that a run could encounter. This path, which may differ from the one in the trace, is considered by the constraint solver to find an input that follows the path, or determine that no such input exists.

An IPG stores a collection of causal relationships. For instance the instruction "mov *eax*, *ebx*", which replaces the value of *eax* with that of *ebx*, has a causal relationship of (*eax* ← *ebx*). During a Database Ingest where (*ebx* ← *ecx*) precedes (*eax* ← *ebx*), the two are linked so the implicit relationship (*eax* ← *ecx*) can be inferred. Thus IPG queries provide crucial taint information.

DBS uses Parse Trace to extract memory accesses along with call and return information for functions. This is used to track the lifetime of heap allocations and stack frames in order to detect the first occurrence of problems such as memory access violations.

5. DATABASES

ARE databases are distributed across multiple computers and between both NoSQL and SQL databases. NoSQL databases are preferable for the data generated by dynamic analysis due to their high performance for large yet transient datasets. Though not as mature as traditional SQL databases, they are stable enough for large-scale use by companies such as Facebook and Yahoo.

5.1 NoSQL Database

Apache's Hadoop offers an efficient file system for large datasets across a cluster of machines [5]. In particular, it supports a MapReduce framework for parallel processing across an extensible number of cluster nodes. Apache's HBase provides a convenient interface to this file system for NoSQL storage and retrieval. This enables distributed processing of the large execution traces that may be generated during dynamic analysis.

5.2 SQL Database

Information from static analysis components, such as the ARE preprocessor and TMF, are stored in SQLite databases for compactness and the ability to use rich SQL queries. The TMF database contains tables named *Instruction*, *Block*, *Data*, *Loop*, *Function*, and *Segment*. *Instruction* holds parsed PREIL instructions. *Block* contains rows of block addresses and conditions for branching from a block. *Data* holds raw TMF information for each block that is aggregated during queries. *Loop* and *Function* hold aggregations for some loops and functions respectively. *Segment* stores descriptions for all functions.

5.3 Database Ingest

Database Ingest involves storing the output of static and dynamic analyses. For example during taint analysis, Trace Parser processes each instruction record independently. This is amenable to MapReduce frameworks where parallel processing is efficient since communication among tasks is not allowed. When *n* task processors are available, an instruction sequence is divided into *n* groups that are processed in parallel. Once the IPG is built, the database establishes taint flows that seamlessly span the groups.

6. BUG SEARCH

ARE combines static and dynamic bug searches. For example, a static bug search that considers the entire application space can locate potential bugs to help prioritize and limit the number of promising traces. Then, a dynamic bug search that considers one trace at a time can check paths to offer high assurance.

6.1 Static Bug Search

TMF abstracts binary applications into algebraic expressions that represent application behaviors and form searchable patterns. Hunting for bugs can therefore be done systematically by finding patterns of bad, dangerous, or otherwise interesting behaviors.

ARE's Static Bug Search (SBS) currently supports bugs such as buffer overflow and double free. Patterns for buffer overflow include data transfer loops whose exit conditions are controllable by user inputs. Patterns for double free include consecutive frees of equivalent pointer symbols. Note that despite its name, SBS is not limited to finding bugs. Other behaviors of interest can be specified via a convenient, high-level API.

6.2 Dynamic Bug Search

ARE's Dynamic Bug Search (DBS) supports stack and heap based memory errors by tracking stack frame setup and maintaining a map of all allocations and deallocations that occur within a trace. DBS checks all memory accesses to detect if any are out of bounds or scope. Stack based bugs include return address overwrite and uninitialized memory. Heap based bugs include buffer overflow, buffer underflow, double free, use after free, and uninitialized memory. DBS can also detect bugs unrelated to memory such as privileged accesses and authentication bypasses.

7. CONSTRAINT SYSTEM

ARE's Constraint Optimization, Management, Extensions, and Translation System (COMETS) establishes constraints matching the weakest preconditions for a given path. Thus, for a deterministic application, a solution to these constraints corresponds to an input value assignment that would drive execution to follow the

given path. Moreover, if the constraint solver properly terminates without finding a solution, then there is no input assignment that can make execution follow the given path. This path is specified by CSI and terminates at an area of interest such as an unexplored instruction or the location of a potential bug, according to SBS.

The "O" in COMETS pertains to optimizing the constraint program, e.g., by reducing its size and complexity so it can be solved in a reasonable amount of time. Strategies for cutting out unnecessary constraints include Statically Limited Irrelevant Constraint Elimination (SLICE), Dynamic Irrelevant Constraint Elimination (DICE), and TMF Exclusions for Augmented Reduction (TEAR). SLICE removes constraints for PREIL instructions that no path can execute, e.g., a write that is always overwritten before it is read. DICE removes constraints that do not affect the set of solutions, e.g., a write that is not read in the given trace. TEAR removes constraints for branches that, according to TMF, do not affect the relationship between inputs and outputs, e.g., a branch that does not influence whether paths reach the point of interest.

The "M" in COMETS pertains to managing services such as joining subproblems (e.g., when long paths are partitioned into less demanding slices) and queries such as finding solutions that maximize some input values while minimizing others (e.g., to find the shortest input that causes a buffer overflow).

The "E" in COMETS pertains to extending the constraint program by further restricting inputs (e.g., to prefer ASCII strings), outputs (e.g., to ensure a value falls within a desired range), and interim variables (e.g., to check constraints around dangerous code).

The "T" in COMETS pertains to translation to and from one or more SMT constraint solvers for fixed-size arrays of bits. Third-party solvers are encapsulated to ensure separation of concerns.

8. ARE OPERATIONS

This section describes how ARE components work together to perform various operations on machine code. For efficiency, these operations can be performed in parallel.

8.1 Dynamic Analysis

Target applications may be executed in the controlled environment of a simulator (§4.1). An initial run with arbitrary inputs can supplement static analysis with information about the initial machine state. Later runs use inputs provided by COMETS to either drive execution toward areas of interest or verify the presence of potential bugs (§7). These runs produce traces that Trace Parser filters and sends to CSI for constraint solving, IPG for taint analysis, and DBS for bug detection (§4.2). Dynamic analysis results are stored in HBase (§5.1).

8.2 Static Analysis

The ARE preprocessor can statically analyze a target application with or without a trace (§3.1). If provided, traces provide data that is only available at runtime such as initial register and memory values and the addresses of external functions. The preprocessor translates native instructions to simple PREIL instructions (§3.2). TMF then transforms the PREIL to algebraic expressions which are aggregated to capture the relationship between inputs and outputs at the instruction, basic block, and function levels (§3.3). The aggregated expressions are stored in SQLite (§5.2).

8.3 Static Bug Search

SBS queries the SQLite database to identify potential problems such as buffer overflows and double frees (§6.1). Each query looks for a generic pattern that may be indicative of a bad behavior. When TMF has fully populated the database, these searches cover the entire application space. SBS outputs a list of matches along with rationales and instruction addresses.

8.4 Guided Path Search

Using the path specified by a CSI trace, COMETS solves for an input that is applied in another iteration of dynamic analysis (§7). Each iteration visits a new path that is selected with a preference for flows that approach the instruction addresses returned by SBS.

8.5 Dynamic Bug Search

To detect memory errors in a trace provided by Trace Parser, DBS finds all calls and returns as well as all memory load and store operations (§6.2). This is used to chart the lifespan of heap and stack allocations. For scalability, DBS employs MapReduce to create and store these allocation maps in HBase. It then steps through all relevant memory accesses to detect if a bug occurred. The output is a report of memory-based bugs that were discovered including memory access violations, double free, use after free, uninitialized memory, and return address overwrite. DBS uses a similar approach to detect other types of bugs as well.

8.6 Parallel Processing

Though there are some ordering constraints, ARE operations can be executed in parallel. For example, since each trace is stored independently in the database, Dynamic Bug Search can examine a trace whilst Dynamic Analysis generates a new trace using a different input. Also, when multiple systems are simultaneously performing Dynamic Analysis, their results are stored by parallel MapReduce jobs. Note that a single set of Static Analysis outputs can be reused by multiple Dynamic Analysis runs.

9. CONCLUSION

Given only a target application, without its source code or other specifications, ARE can locate the addresses of potential bugs, identify execution inputs that reach those addresses (if any), and determine whether or not the bugs can be triggered. ARE also offers other analytics, e.g., to indicate which registers and memory locations need to be controlled in order to express the bugs.

10. REFERENCES

[1] Brumley, D., Jager, I., Avgerinos, T., and Schwartz, E. J. 2011. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Snowbird, UT, July 14-20, 2011), 463-469.

[2] Lim, J. and Reps, T. 2013. TSL: A System for Generating Abstract Interpreters and its Application to Machine-Code Analysis. *ACM Trans. Program. Lang. Syst.* 35, 1, Article 4 (April 2013), 1-59.

[3] Ross, R. 2013. ARE: A System for Automated Reverse Engineering. In *Poster Session of the 13th Annual High Confidence Software and Systems Conference* (Annapolis, MD, May 7-11, 2013).

[4] Dullien, T. and Porst, S. 2009. REIL: A Platform-independent Intermediate Representation of Disassembled Code for Static Code Analysis. In *CanSecWest* (Vancouver, Canada, March 12-14, 2009).

[5] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. 2010. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies* (Incline Village, NV, May 3-7, 2010), 1-10.

Appendix A. ARE DEMONSTRATION

This section illustrates the steps taken by ARE when it analyzes machine code for an application and discovers a buffer overflow.

A.1. Preprocessor

ARE preprocessing consists of the nine steps shown in Figure 2. Steps 1, 2, and 3 prepare the environment and consult IDA Pro. Step 4 reads an execution trace if one is provided. Steps 5, 6, and 7 collect all static instructions in the application and reconcile them with every instruction that was observed in the trace. Steps 8 and 9 translate native instructions to PREIL and output the static information. For the example in Figure 2, dynamic analysis of the application was not performed so no trace was available.

```
---------- Step 1: Clean up all leftover files from previous runs
---------- Step 2: Parse map to find library names and addresses
---------- Step 3: Read library source files into IDA
Processing file 001 of 001 -- _AT_0x00400000_bof2.exe
---------- Step 4: Parse execution trace
---------- Step 5: Produce static instructions from library source files
Processing file 001 of 001 -- _AT_0x00400000_bof2.exe
---------- Step 6: Check static instructions against execution trace
---------- Step 7: Adjust static instructions from execution trace
Processing file 001 of 001 -- _AT_0x00400000_bof2.exe
---------- Step 8: Produce final instructions, PREIL, block, and other files
Processing file 001 of 001 -- _AT_0x00400000_bof2.exe
Processing blocks
Gathering files
Verifying REIL output
---------- Step 9: Final reconciliation / finalization
```

Figure 2. ARE preprocessing log

A.2. TMF

The TMF processor scans all instructions in the preprocessor output and builds expressions for each basic block. These expressions are stored in an SQLite database. Figure 3 displays the TMF expressions for the basic block at address 0x4003c0. Here, the most common activity is saving a register to the stack.

```
sqlite> select * from data where addr='0x4003c0';
0x4003c0|[(0x14+($sp+0xfffffff8))] = $fp|0x0
0x4003c0|$fp = ($sp+0xfffffff8)|0x4003c8
0x4003c0|[(0x18+($sp+0xfffffff8))] = $a0|0x0
0x4003c0|[(0x1c+($sp+0xfffffff8))] = $a1|0x0
0x4003c0|[(0x8+($sp+0xfffffff8))] = $zero|0x0
0x4003c0|$sp = ($sp+0xfffffff8)|0x4003c0
```

Figure 3. TMF expressions for a basic block

A.3. Static Bug Search

Figure 4 shows how SBS identified the instruction address of a potential buffer overflow bug. First, the pattern matcher found a buffer copy loop at address 0x400408. Here, the TMF expression *"[[$fp+0x8] + [$fp+0x18]] = 0x61 & 0xff"* was recognized as a data transfer with *"[$fp+0x8]"* as its index. Next, SBS found that the exit condition of this loop was regulated by two memory locations, *"[$fp+0x1c]"* and *"[$fp+0x8]"*, which corresponded to two arguments, a buffer pointer *$a0* and a number of bytes *$a1* respectively. Since an input argument matched the index of a write to memory, SBS alerted that *$a1* (the number of bytes to write) controlled a potential buffer overflow bug at 0x400408.

```
User controlled exit condition found
Loop: '0x400408  0x4003e0' has buffer copying
Data transfer expression:
[([($fp+0x8)]+[($fp+0x18)])] = ((0x61)&(0xff))
Index = [$fp+0x8]
Exit condition: ((((((([($fp+0x8)])&(0x80000000))))>(0x1f))^(
((([($fp+0x1c)])&(0x80000000))))>(0x1f)))&((((([($fp+0x8)])&(0
x80000000))))>(0x1f))^(((([($fp+0x8)]-[($fp+0x1c)])&(0x100000
000)))))>(0x20))))^(((([($fp+0x8)]-[($fp+0x1c)])&(0x100000000)
)))>(0x20))) == 1
Inputs from exit condition: [($fp+0x1c)],[($fp+0x8)]
Controllable input: $a1
```

Figure 4. Potential buffer overflow identification

A.4. Simulators

The addresses returned by SBS are points of interest that ARE wants execution to reach, and the inputs returned by SBS are variables that ARE wants to control, but ARE do not originally know what application inputs will accomplish these goals. Thus, ARE starts by running a simulator with arbitrary inputs to capture an initial state description and produce an initial trace. A sample trace is shown in Figure 5. Here, the start of the basic block mentioned in Section A.2 is highlighted. Traces for other inputs can be created by either a simulator or the Emulated Tracer.

```
160|1|0x4005e0||move $t9, $v0|||R:$v0:0x4003c0|W:$t9:0x4003c0|
161|1|0x4005e4||bal fill_buffer;nop||||W:$ra:0x4005ec|
162|1|0x4003c0||addiu $sp, -0x18|||R:$sp:0x7fbc|W:$sp:0x7fa4|
163|1|0x4003c4||sw $fp, 0x18+var_4($sp)||W:ss:0x7fb8:4:0x7fbc|
164|1|0x4003c8||move $fp, $sp|||R:$sp:0x7fa4|W:$fp:0x7fa4|
165|1|0x4003cc||sw $a0, 0x18+arg_0($fp)||W:ss:0x7fbc:4:0x247d9
166|1|0x4003d0||sw $a1, 0x18+arg_4($fp)||W:ss:0x7fc0:4:0x14|R:
167|1|0x4003d4||sw $zero, 0x18+var_10($fp)||W:ss:0x7fac:4:0x0|
```

Figure 5. Excerpt from a trace

A.5. Database Ingest

Data extracted from traces are stored in HBase. Each ingest process is a MapReduce job. Figure 6 displays their statuses.

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete
job_201402201925_0001	NORMAL	user	importtsv_bof2.mips.02.18.2014_module	100.00%	1	1	100.00%
job_201402201925_0002	NORMAL	user	importtsv_bof2.mips.02.18.2014_function	100.00%	1	1	100.00%
job_201402201925_0003	NORMAL	user	importtsv_bof2.mips.02.18.2014_block	100.00%	1	1	100.00%
job_201402201925_0004	NORMAL	user	importtsv_bof2.mips.02.18.2014_instruction	100.00%	1	1	100.00%
job_201402201925_0005	NORMAL	user	importtsv_bof2.mips.02.18.2014_instr_reil	100.00%	1	1	100.00%

Figure 6. Job statuses

A.6. Constraint System

COMETS creates and solves constraints based on trace data in order to find inputs that will drive execution toward the points of interest. An example solution is shown in Figure 7. Solutions map to inputs that lead to new paths for simulation and HBase ingest.

```
Solution:
 main[0x7FF4] = 0x00
 main[0x7FF8] = 0x00
 main[0x3180E1D] = 0x32
```

Figure 7. Constraint solution

A.7. Dynamic Bug Search

DBS reads trace data and, when pursuing memory bugs, updates a memory table based on the sequence of allocations and deallocations in the trace. For example, a memory access violation detector checks the current table at each memory read or write to ensure the location is accessible. Figure 8 shows a confirmed buffer overflow bug that was triggered by the instruction at address 0x4003f4. Specifically, addresses 0x247d9e70 through 0x247d9e75 were written at a time when they were not allocated.

memaccess_violation.log

```
1 Seq#|Read/Write|InstrAddr|MemoryAddrAccessed
2 329|WRITE|4003f4|247d9e70
3 344|WRITE|4003f4|247d9e71
4 359|WRITE|4003f4|247d9e72
5 374|WRITE|4003f4|247d9e73
6 389|WRITE|4003f4|247d9e74
7 404|WRITE|4003f4|247d9e75
```

Figure 8. Detected memory access violations