REIL: A platform-independent intermediate representation of disassembled code for static code analysis

Thomas Dullien
zynamics GmbH
Bochum, Germany
thomas.dullien@zynamics.com

Sebastian Porst
zynamics GmbH
Bochum, Germany
sebastian.porst@zynamics.com

ABSTRACT

In this paper we introduce REIL (Reverse Engineering Intermediate Language), a platform-independent intermediate language to represent disassembled assembly code. We created the REIL language specifically to simplify and automate static code analysis of assembly code in the context of software reverse engineering for the purpose of security auditing and vulnerability detection.

This paper introduces the complete REIL language with all of its instructions as well as the virtual REIL architecture that defines the effects of REIL instructions on registers and memory. Furthermore we discuss the reasons why we designed the REIL language the way we did, what limitations the user of the language should be aware of, and what we have planned for REIL in the future.

Keywords

REIL, Reverse Engineering Intermediate Language, Static Code Analysis, Disassembly, Intermediate Representation, Intermediate Language Recovery

1. INTRODUCTION

Only a few years ago the main exposure people had to security-critical computer programs like credit card stealing malware came through their home computers. Due to the dominance of Microsoft's operating system Windows these computers were nearly always computers of the x86 family. This situation changed. People today come in contact with more and more computer architectures that are directly or indirectly relevant to the safety of their private data. Examples include appliances like modern cell phones, network printers with integrated web servers and hard drives, and more complex routers or wireless devices that are now part of many home networks.

On the side of the security researchers this led to a diversification of target architectures that need to be analyzed. Of course there is still the x86 platform which is the bread and butter of many security researchers but other devices use different CPUs. In the mobile world ARM is the most popular architecture, while network appliances like routers often use PowerPC CPUs. MIPS is also making a comeback in wireless devices and some high-end routers.

For a typical security research and consulting company that wants to diversify its product palette this creates a need to have tools that can work on assembly code of several different platforms. Here is where our contribution comes in. We developed an analysis language called REIL (Reverse Engineering Intermediate Language) that abstracts from native assembly code and therefore makes it possible to develop analysis tools and algorithms that work on many different platforms.

One thing that sets REIL apart from other proposed analysis languages is that REIL is not just a prototype language. A REIL implementation was already shipped in a commercial reverse engineering tool (BinNavi) where it has proven to be very valuable in developing new static analysis algorithms in real-world software analysis scenarios.

2. THE REIL INSTRUCTION SET

One of the most important advantages of the REIL language is its very reduced instruction set. REIL knows only 17 different instructions. This distinguishes REIL significantly from all popular instruction sets supported by real CPUs today. For example, the x86 instruction set including all of its modern extensions contains more than 600 instructions. The PowerPC instruction set including all simplified mnemonics contains more than 1000 instructions[1]. The reduction to a core minimum of REIL instructions was deliberately chosen to make it as easy as possible to write static analysis algorithms for REIL code. The idea behind this is that fewer instructions in an assembly language mean fewer different transformations of program state that must be considered in a static analysis algorithm.

Another advantage of REIL instructions over instructions of common architectures is the single-responsibility aspect of all REIL instructions. Typical instructions of real architectures often have many different responsibilities. A single x86 instruction can load a value from memory, perform some kind of computation on it, and set flags according to the result of the computation. In REIL this is not possible. Every instruction has exactly one effect on the program state. Either it loads a value from memory, or it performs a

[1]Simplified mnemonics are relevant here because in nearly all cases where binary programs are disassembled for security analysis, industry standard disassemblers like IDA Pro generate simplified mnemonics.

CanSecWest 2009 Vancouver, Canada

computation on a value, or it sets a single flag. This makes it very easy for the user of REIL code to understand what exactly is going on in a REIL instruction. Non-obvious side-effects that require a deep understanding of the underlying instruction set can never happen.

The operands of REIL instructions are also very regular. Each REIL instruction has exactly three operands. The first two operands of a REIL instruction are in all cases input operands which are never modified by the instruction. The third operand is generally the output operand of the instruction. This operand stores the result of the computation performed by the instruction[2]. Not all instructions need to have three different operands. The most obvious example is the no-operation instruction NOP which does not need to have any operand. Nevertheless, the REIL instruction NOP still has three different operands of type *Empty*.

Except for *Empty*, there are only three more types of REIL instruction operands. Operands can be integer literals, registers, or subaddresses. Of those three operand types, integer literals are the simplest type. Operands of this type are typically used when a constant integer value like 5 or 4379 is required as part of the computation of a more complex result. Since integer literals are read-only they can never be used as output operands of REIL instructions.

Registers are the second type of REIL operands. REIL registers work exactly like native general-purpose CPU registers. They can hold integer values and they are mutable, meaning the value they hold can be changed.

In fact, while most registers used in REIL instructions are pure REIL registers of the form $t_{<number>}$ (e.g. t_0, t_1, ... , t_{123}, ...), native registers from the source architecture of an analyzed program can also show up as operands of REIL instructions. This does not limit the platform-independence of REIL code as REIL registers and source architecture registers are treated completely uniformly in REIL analysis algorithms. Native registers are simply used in REIL code to make it easier to liken the results of an analysis algorithm back to the original input code.

The last REIL instruction operand type is the subaddress. Operands of this type are comparable to integer literals but instead of integral values these operands always hold addresses of REIL instructions. Furthermore, this operand type can only appear as the third operand of JCC (conditional jump) instructions. Operands of this type are only generated when an original native assembly instruction is translated into a series of REIL instructions that contains branches from decisions or loops. Examples for such instructions are the prefixed string operations (rep stos, ...) of the x86 instruction set which are translated to REIL instructions that form a loop.

Except for their type and their value, REIL operands are characterized by their size. This size is equal to the maximum size of its operand value. REIL operand sizes have names like b_1, b_2, b_4 and so on meaning that the size of the operand is 1 byte, 2 bytes, and 4 bytes respectively. For example, the integer literal operand $0x17/b_2$ is really two bytes large and could also be represented as $0x0017b_2$ while the size of the register t_0 in the operand t_0/b_4 is 32 bits.

In addition to its operands, all REIL instructions can come with so-called meta-data. This meta-data is simply a map of key-value pairs that give additional information about an instruction that is probably important during static code analysis. In general, the number of pieces of meta-data associated with an instruction is not limited but in practice most REIL instructions do not have any meta-data at all associated with them.

In the current version of REIL there is only one kind of meta-data. Jump instructions that were generated during the translation of a subfunction call (like *call* on the x86 CPU or *bl* on PowerPC) are specifically marked with the key *isCall* and the value *true*. This is necessary because subfunction calls need to be treated very differently than conditional jumps during many static code analysis algorithms.

The 17 different REIL instructions can be grouped into a few different instructions groups. The biggest group are the arithmetic instructions like addition and subtraction. Then there are the bitwise instructions that perform operations like bitwise OR and AND, the conditional instructions that are used to compare values and jump according to the result of the comparison, the data transfer instructions that access REIL memory and transfer the content of registers, and the remaining instructions which do not really fall into any group.

2.1 The arithmetic instructions

With six members, the group of arithmetic instructions covers more than one third of the total instructions of the REIL instruction set.

- ADD: Addition of two values
- SUB: Subtraction of two values
- MUL: Unsigned multiplication of two values
- DIV: Unsigned division of two values
- MOD: Unsigned modulo of two values
- BSH: Logical shift operation

ADD and SUB work exactly like standard addition and subtraction on most platforms.

The multiplicative instructions MUL, DIV, and MOD interpret all of their input operands in an unsigned way. The REIL instruction set does not contain signed counterparts of these instructions because signed multiplication and division can easily be simulated in terms of unsigned multiplication and division.

The logical shift operation can either be used as a left shift or a right shift, depending on the sign of its second operand. If the second operand is positive, the shift operation is a left-shift. If it is negative, the shift operation is a right-shift. Arithmetic shifts do not exist in the REIL instruction set because arithmetic shifts can easily be simulated with the help of logical shifts. Like in the case of the multiplicative instructions, keeping the REIL instruction set small was more important than adding the convenience of having more epxressive REIL translations.

Figure 1 shows examples of all arithmetic instructions. The structure of all arithmetic instructions is the same. The first two operands are the input operands of the operation while the third operand is the output operand where the result of the operation is stored. The order of the input operands is the natural order that is generally used when

[2]The one exception is the jump instruction JCC where the third operand is the jump target.

writing down the operations in infix notation on paper or in the source code of computer programs. For example, the first operand of the SUB operation is the minuend while the second operand is the subtrahend. In the DIV operation the first operand is the dividend and the second operand is the divisor.

```
ADD t0/b4,    t1/b4, t2/b8
SUB t7/b4,    t9/b4, t12/b8
MUL t8/b4,    4/b4,  t9/b8
DIV 4000/b4,  t2/b4, t3/b4
MOD t8/b4,    8/b4,  t4/b4
BSH t1/b4,    2/b4,  t2/b8
```

Figure 1: Examples of the arithmetic REIL instructions

Another important aspect of REIL is first shown in figure 1 too. Potential overflows in the results of operations are handled explicitly. If an operation can overflow, the output operand must be large enough to store the whole result including the overflow. This is the reason why the output operands of the example instructions are twice as large as their input operands[3]. The two exceptions are the output operands of the DIV and MOD instructions. Since the results of these operations can never be larger than the first input operand, an extension of the size of the output operand is not necessary. The output operand has the size of the input operand instead.

The explicit handling of overflow is an important difference to real architectures where overflows produced by operations are nearly always cut off because of the fixed size of native CPU registers. This explicit overflow handling is what enables REIL algorithms to analyze the results of operations in bigger detail when the exact overflowing value of a register might be important instead of simply having a flag that signals that an operation produced an overflow.

2.2 The bitwise instructions

The next biggest instruction group is the group formed by the three bitwise instructions.

- AND: Bitwise AND of two values
- OR: Bitwise OR of two values
- XOR: Bitwise XOR of two values

The three bitwise instructions work exactly like one expects bitwise instructions to work. Bit for bit they connect the bits of two input operands according to the truth table defined for their operation. The calculated value is then written to the output operand of the instruction.

A bitwise NOT instruction is not part of the REIL instruction set because NOT is equivalent to XOR-ing a value with a value of equal size and all bits set. That means to calculate the one's complement of the 16-bit value 0x1234 one would XOR it with the 16-bit value 0xFFFF.

[3]The result operand of addition and subtraction is technically too large because these operations performed on two 32-bit values can only overflow into the 33rd bit; however there is no 33-bit REIL operand size so the next biggest operand size (64-bit) was chosen.

```
AND t0/b4,    t1/b4, t2/b4
OR  t7/b4,    t9/b4, t12/b4
XOR t8/b4,    4/b4,  t9/b4
```

Figure 2: Examples of the bitwise REIL instructions

Figure 2 shows examples of all bitwise instructions. Their general structure equals the structure of the arithmetic instructions. Like them, bitwise instructions take two input operands and store the result of the operation in the output operand. One important difference is that none of the bitwise instructions produce an overflow. An explicit modeling of overflowing values and an extension of the size of the output operand are therefore not necessary.

2.3 Data transfer instructions

To access the REIL memory, two different REIL instructions are needed. One is used for loading a value of arbitrary size from the REIL memory while the other one is used to store a value of arbitrary size to the REIL memory. Furthermore, this group of instructions contains an instruction that is used to transfer values into registers.

- LDM: Load a value from memory
- STM: Store a value to memory
- STR: Store a value in a register

The first operand of the LDM instruction contains the address of the REIL memory where the value is loaded from. This operand can either be an integer literal or a register. When the instruction is executed, it loads the value from the given memory address and stores it in the third operand of the instruction. The size of the value that is loaded from memory equals the size of the third operand. If the size of the third operand is a 32-bit register, a 32-bit value is loaded from memory. As the loaded value is written to the third operand, the third operand must be a register.

The store operation STM is the inverse operation to the load operation LDM. It can be used to store a value of arbitrary size to memory. The first operand of the STM instruction is the value to be stored in memory. Its size determines how many bytes are written to memory when the STM instruction is executed. The third operand specifies the address where the value of the first operand is written to. Both operands can be either integer literals or registers. The second operand is unused.

The STR instruction is one of the simplest instructions of the REIL instruction set. It copies a value to the output register specified in the instruction. The input operand can be either a literal (to load a register with a constant) or another register (to transfer the content of one register to another register).

```
LDM  413800/b4, , t1/b2
STR  t1/b2      , , t2/b2
STM  t2/b2      , , 415280/b4
```

Figure 3: Examples of the data transfer REIL instructions

Figure 3 shows a sequence of data transfer instructions that load a value from memory, copy it to another register,

and store it back to another address in memory. Since the size of the output register of LDM instructions specifies how many bytes are loaded from memory, it is clear that two bytes are loaded from memory. The size of the two used operands of STR instructions is typically the same as STR only copies a value. In the end the two-byte register t_2 is stored back to memory.

2.4 Conditional instructions

The group of conditional instructions is used to compare values and depending on the results of the comparison to jump to one REIL instruction or another.

- BISZ: Compare a value to zero
- JCC: Conditional jump

The BISZ instruction is the only instruction of the REIL instruction set that can be used to compare two values. In fact, it can only be used to compare a single value to zero but this is sufficient to emulate any kind of more complex comparison. The BISZ instruction takes a single operand, compares it to zero, and depending on the value of the input operand, the output operand is set to 0 (if the value of the input operand was not 0) or 1 (if the value of the input operand was 0).

The conditional jump instruction JCC is typically used to process the results of a BISZ instruction. If the first operand of the JCC instruction evaluates to 0, the jump is not taken. If the first operand evaluates to any other value than zero, the jump is taken and control is transferred to the address (or sub-address) specified in the third operand.

An unconditional jump is not part of the REIL instruction set because it is possible to emulate an unconditional jump using a conditional jump by setting the first operand of the conditional jump to the integer literal 1 (or any other non-zero integer literal).

```
BISZ t0/b4, , t1/b1
JCC  t1/b1, , 401000/b4
```

Figure 4: Examples of the conditional REIL instructions

Figure 4 shows a typical sequence of a single BISZ instruction followed by a JCC instruction that uses the output of the BISZ instruction to determine whether to take a jump to the address specified in its third operand. Since the output of BISZ instructions is always either 0 or 1, the size of the output operand of BISZ instructions is always b_1.

2.5 Other instructions

There are a few other instructions which do not really belong to any group at all.

- UNDEF: Undefines a value
- UNKN: Unknown source instruction
- NOP: No operation

The UNDEF instruction undefines the value of a register. This means that once the UNDEF instruction is executed, the value inside the undefined register is unknown. This is important because there are native assembly instructions which leave registers or flags in an undefined state. The x86 instruction DIV, for example, leaves a number of flags like the zero flag and the carry flag in an undefined state.

The UNKN instruction is kind of a placeholder instruction. It indicates that during the REIL code generation an original assembly instruction was encountered that could not be translated.

The NOP instruction does nothing. Nevertheless it is not useless. REIL translators can generate this instruction to pad control flow in certain edge cases. In a few situations this is very useful because it keeps REIL translators very simple. Without the existence of the NOP instruction, the REIL translator would have to look ahead to the next native instruction to generate correct REIL code[4].

```
UNDEF , , t1/b4
UNKN  , ,
NOP   , ,
```

Figure 5: Examples of other REIL instructions

Figure 5 shows examples of the remaining REIL instructions. The only instruction that takes operands is the UNDEF instruction which undefines a register.

3. THE REIL ARCHITECTURE

The definition of the REIL language includes the description of the REIL architecture and the definition of a virtual machine that can be used to execute the generated REIL code.

The REIL architecture is a simple register-based architecture without an explicit stack. The number of registers available in REIL code is unlimited. As previously explained, the names of REIL registers have the form $t_{<number>}$. The index number of register names is unbounded. There is furthermore no requirement that all REIL registers between t_0 and t_{n-1} are used by a given program that uses n different registers. A program that uses exactly three REIL registers can use t_7, t_{799}, and t_{3199} if desired.

REIL registers themselves do not have a fixed width or a limited width. The size of REIL registers is always equal to the size of the operands where they are used. The size of REIL registers can even change between instructions. In one instruction register t_n can have size b_s while in another instruction it can have size b_t. Since operands can grow arbitrarily large, REIL registers can also grow arbitrarily large. In practice we have not yet encountered registers with more than 128 bits (equivalent to b_{16}) though.

We already mentioned that registers of the original input code can appear in REIL code. In fact, the registers of the original architecture will always appear in REIL code to make it possible to port results of REIL code analysis back to the original code. This does not violate the platform-independent nature of REIL code. REIL registers and native registers can be mixed at will and be treated completely uniformly. While analyzing REIL code there is no difference between the registers t_0, t_1, and t_2 and the registers eax, ebx, and ecx. At the end of an analysis algorithm one can then easily distinguish between the REIL registers (which have the t_n form) and the native registers (which do not have the

[4]Technically, the NOP instruction could of course be replaced by an instruction like *add 0, 0, t_n* that also has no discernible effect on the program state.

t_n form) to port the values of relevant registers back to the original assembly code.

The memory of the virtual REIL machine follows a flat memory model. Unlike some real CPUs like the x86 which has memory segments (in real mode) or at least memory selectors (in protected mode), REIL memory starts at address 0 and can grow arbitrarily large. While there is technically an infinite amount of storage available in REIL memory, practical concerns of the source architecture limit the used memory in practice. If the source assembly language (like 32 bit x86 assembly) can only address 4 GB of memory, only 4 GB of REIL memory will ever be accessed in REIL code created from x86 programs. REIL memory higher than the addressable memory range of the source target architecture is never used.

Due to the flat memory model of the REIL memory, segmented memory access of native architectures must be simulated in REIL programs if necessary. This can be done by creating virtual segments which represent the memory segments of the native architecture. Since REIL memory is not limited in size, there is enough space available to make these virtual segments non-overlapping, meaning that memory access through one segment of the native architecture never interfers with memory access through another segment of the native architecture.

The endianness of the source architecture must be considered too when accessing REIL memory. On native architectures endianness falls into two different categories. In some cases (like x86) native architectures have a fixed endianness that can not be changed during runtime while other architectures can switch the endianness of their memory access at runtime by executing a special instruction (PowerPC, ARM). In general, REIL does not have any mechanisms to deal with endianness. All endianness issues must be handled by the REIL translators when generating the REIL instructions that access memory. This poses a problem when endianness is switched at runtime because REIL code is generated in advance and can not be updated anymore when endianness-switching happens. However, the rarity of endianness-switching makes this a special situation that is seldomly relevant for security audits.

After REIL memory and the REIL registers are given an initial state, REIL code can be analyzed or even executed. Execution of REIL code happens just like program execution on a real CPU. Starting with the value in the program counter register, REIL code is executed[5]. The REIL instruction at the position of the current program counter is fetched and interpreted with regard to the current state of the REIL register bank and the REIL memory. Once interpretation is complete, the REIL register bank and the REIL memory are updated to reflect the effects of the instruction on the global state.

4. TRANSLATING NATIVE CODE TO REIL

The translation of native assembly code to REIL code is straightforward. For each supported native assembly language there is a so called REIL translator. This REIL translator takes a piece of native assembly code and translates it to REIL code. Linearly iterating over all instructions in a piece of input code, the translator translates each instruction to REIL code independently. The REIL translator does not look ahead to see what instruction follows the current instruction and it does not require information generated during the translation of previous instructions. This statelessness of the translation makes REIL translators very simple. In fact, REIL translators are nothing but glorified maps that repeatedly map a single native instruction to a list of REIL instructions.

Due to the simplicity of REIL instructions and what they can do in one step, a single native assembly instruction is nearly always translated to many REIL instructions. Experimental results have shown that on average, an original instruction is translated into approximately 20 REIL instructions while the most complex native instruction we found in practice was translated to more than 50 REIL instructions.

This one-to-many relation between native instructions and REIL instructions unfortunately destroys a direct correspondence between the address of a native assembly instruction and the addresses of the REIL instructions created for the native assembly instruction. Having such a correspondence would be most desirable because it would make it significantly simpler to port the results of a REIL analysis algorithm back to the original assembly code. To solve this problem, the addresses of REIL instructions are shifted to the left by 8 bits (or multiplied by 0x100). This means that the first REIL instruction that corresponds to the native assembly instruction at offset n has the offset $0x100 * n$ while the second REIL instruction has the offset $0x100 * n + 1$ and so on. This address translation limits the translation of a single native instruction to at most 256 different REIL instructions. Should it ever happen that more than 256 REIL instructions are generated for a single native instruction, the addresses of the REIL instructions would overflow into the addresses of the REIL instructions of the following native instruction.

5. LIMITATIONS OF REIL

There are a number of more or less significant issues that might limit the use of REIL in practice. Some of these limitations are built into the REIL language itself while others exist simply because we have not yet had time to implement certain aspects of native architectures.

The first limitation is that the REIL translators we have so far (32-bit x86, 32-bit PowerPC, and 32-bit ARM) are unable to translate certain classes of instructions. For example, none of the translators can translate FPU instructions. CPU extensions like the MMX and SSE extensions of x86 CPUs are also not translated yet. We have chosen to skip the translation of these instructions because REIL is supposed to be a language for analyzing assembly code for security-critical bugs and vulnerabilities. FPU, MMX, and SSE extensions are only very rarely involved in these kinds of flaws. Should FPU bugs or other CPU extension bugs become popular targets of software exploits in the future, we can easily extend our existing translators to be able to handle these instructions.

Like FPU instructions, privileged instructions like system calls, interrupts, and other kernel-level instructions are not translated by our current REIL translators. The justification for the lack of support for these kinds of instructions

[5]There is no special REIL program counter register. Rather, the program counter register of the input architecture is used. This is important to make sure that at each step of the REIL code analysis, the value of the program counter register has the same value as it would have during a real execution of the program on the source platform.

follows along the lines of the lack of FPU support. In our initial implementation of REIL we wanted to focus on the instructions that are most often involved in some kind of security-relevant software flaws. Depending on the exact effects of the missing privileged instructions, it might be trivial to impossible to add them to the REIL language. An instruction that has significant low-level effects on the underlying hardware, for example one that flushes the CPU cache, will never be part of REIL for this would mean a complete loss of platform-independence and/or a big increase in the number of different instruction mnemonics. Other privileged instructions like interrupt execution can often be simulated using the features REIL already has.

REIL can also not deal with exceptions in a platform-independent way. This means that at this point exceptions and the corresponding stack unwinding can not be handled by REIL. Due to the lack of exception handling common situations that throw exceptions (dividing by zero, hitting a breakpoint, ...) are simply ignored in the default REIL interpreter.

The next limitation is that REIL can not handle self-modifying code of any kind. This is simply because native code is pre-translated instruction for instruction of a native function and the resulting REIL code is fixed after the initial translation. The reason for this is that REIL instructions themselves do not reside in the REIL memory. They can therefore not be overwritten and modified during the interpretation of REIL code.

6. THE FUTURE OF REIL

The first and foremost goal of the next few months is to write more REIL translators (for example to translate MIPS code) and to implement more REIL-based code analysis algorithms. Additionaly, we have a few minor ideas about improving the quality of generated REIL code and its usefulness in static code analysis.

The first idea is the introduction of a bit-sized operand type b_0. Right now the smallest operand type is the byte-sized operand b_1. During bit-width analysis it might be useful to know that an operand that has size b_1 in current code does not use any bits but its least significant bit. Extending on this idea, maybe it would be smarter to give the size of operands in bits instead of bytes.

An idea that can be used to improve the correctness of REIL translation and certain analysis algorithms is the introduction of two additional instructions, extend and reduce. The motivation for these two instructions is simple. Right now there are no limitations on how operand sizes can be combined in one instruction. When generating an ADD instruction one input operand can have size b_1 while the second input operand can have size b_4. A rule that specifies that the input operands of all instructions must be of equal size would make REIL code more regular for analysis and certain bugs classes in REIL translators can be checked for automatically. The role of the extend instruction would be to extend a value of a smaller size like b_1 to a larger size like b_2 or b_4 while keeping the value of the extended register the same. The reduce instruction would be the opposite of the extend instruction. Reduce would reduce the size of an operand to a smaller operand size. In this case it can not be guaranteed that the value of the reduced register equals the value of the original register. In many situations overflowing high bits will be truncated and lost. This is perfectly acceptable though because this is used in many different situations already, for example when writing the 33-bit wide result of an addition of two 32-bit values back to a 32-bit register while truncating the overflow. Right now this truncation is done using an AND instruction. In the future the reduce instruction might make things semantically clearer.

The number of operand types might also be increased in the future. As soon as FPU instructions are supported by the REIL translators it is necessary to add single-precision FPU operands and double-precision FPU operands. Another example are certain architectures like PowerPC where registers can be addressed not by name but by an index into the register bank. These instructions can not be translated to REIL yet because REIL does not know an operand type like register index.

7. RELATED WORK

The use of intermediate languages for code analysis is nothing new. In fact all serious compilers use some kind of intermediate language during the optimization phase of their generated code (see GCC for example). Creating intermediate representations for disassembled assembly code in the context of security analysis is not nearly as widespread. Nevertheless there are a few approaches which are noteworthy.

At the hack.lu conference 2008 Mihai Chiriac of the antivirus software company BitDefender presented an intermediate language that he used to speed up the emulation of obfuscated malware programs [1]. The intermediate language he presented is structurally close to REIL. Like REIL, his language has a very reduced instruction set where every instruction has exactly one effect on the global state. Furthermore his virtual architecture has an infinite number of virtual registers and a fully emulated memory.

An open-source implementation of an intermediate language specifically made for reverse engineering and statically analyzing binary code is the ELIR language of the ERESI project[6]. Like REIL, the goal of the ELIR intermediate language is simplified platform-independent reasoning about assembly code by providing an intermediate language that makes the effects of all native assembly operations explicit. An overview of the ELIR language was given in Julien Vanegue's EKOPARTY 2008 talk *Static binary analysis with a domain specific language*[2].

A commercial use of intermediate language recovery from disassembled code in the context of security analysis is IDA Pro and Hex-Rays. IDA Pro is the industry standard disassembler for many platforms and Hex-Rays is a decompiler plugin for IDA Pro. The Hex-Rays decompiler uses an intermediate language representation (IR) of the underlying disassembled code to analyze and optimize the disassembled code and to decompile it into a C-style high-level language. As shown in Ilfak Guilfanov's Black Hat 2008 presentation *Decompilers and Beyond*[3] [4], the intermediate representation used by Hex-Rays is significantly different from REIL. There are more instructions in the Hex-Rays IR and they do not obey the single-responsibility rule for avoiding side effects. Other differences include the distinction between integer literals and pointers to code which is present in the Hex-Rays IR but not in REIL and features like the option to address basic blocks instead of addresses in jump instruc-

[6]http://www.eresi-project.org

tions. Another striking difference that can be seen directly when looking at code snippets of REIL and the Hex-Rays IR is that REIL uses way more temporary registers to translate a typical piece of code.

Another implementation of an intermediate language was created by GrammaTech in their CodeSurfer/X86 product. While not publicly available at this point, several whitepapers have been released about CodeSurfer/X86 (for example see [5] or [6]). Unfortunately these whitepapers focus on the results of certain static analysis algorithms with CodeSurfer/X86 instead of their intermediate language so it is unclear at this point how similar this language is to REIL.

As part of AbsInt, an analysis framework specifically suited for statically analyzing embedded system code, Saarland University developed the intermediate language CRL2. Like REIL, CRL2 is generated by transforming the assembly code of a disassembled input program. Nevertheless the similarities to REIL end at this point. CRL2 was specifically developed for detailed control flow analysis and as a result of that, CRL2 code is very complex due to a large number of annotations that are relevant for control flow. Examples of generated CRL2 code can be found at [7].

8. CONCLUSIONS

Using the information presented in this paper it is possible to write a complete implementation of the Reverse Engineering Intermediate Language REIL that can be used for static code analysis of disassembled assembly code. We have already created a commercial implementation of REIL in our product BinNavi and we have successfully written several simple static code analysis algorithms. Thanks to REIL these algorithms work platform-independently on x86 code, on PowerPC, and on ARM code.

9. REFERENCES

[1] Mihai G. Chiriac. Anti Virus 2.0 - Compilers in disguise . hack.lu, October 2008.

[2] Julien Vanegue. Static binary analysis with a domain specific language. EKOPARTY 2008, October 2008.

[3] Ilfak Guilfanov. Decompilers and beyond. BlackHat USA 2008, August 2008.

[4] Ilfak Guilfanov. Decompilers and beyond - Whitepaper . BlackHat USA 2008, August 2008.

[5] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. Codesurfer/x86-a platform for analyzing x86 executables. In *of Lecture Notes in Computer Science*, pages 250–254. Springer, 2005.

[6] T. Reps, G. Balakrishnan, J. Lim, and T. Teitelbaum. A next-generation platform for analyzing executables. In *In APLAS*, pages 212–229, 2005.

[7] AbsInt Angewandte Informatik GmbH. CRL Version 2 Manual .