Data Flow Analysis of Embedded Program Expressions

Christopher Doble **Colin J. Fidge** **Diane Corney**

Faculty of Science and Technology,
Queensland University of Technology, Brisbane

Abstract

Data flow analysis techniques can be used to help assess threats to data confidentiality and integrity in security-critical program code. However, a fundamental weakness of static analysis techniques is that they overestimate the ways in which data may propagate at run time. Discounting large numbers of these false-positive data flow paths wastes an information security evaluator's time and effort. Here we show how to automatically eliminate some false-positive data flow paths by precisely modelling how classified data is blocked by certain expressions in embedded C code. We present a library of detailed data flow models of individual expression elements and an algorithm for introducing these components into conventional data flow graphs. The resulting models can be used to accurately trace byte-level or even bit-level data flow through expressions that are normally treated as atomic. This allows us to identify expressions that safely downgrade their classified inputs and thereby eliminate false-positive data flow paths from the security evaluation process. To validate the approach we have implemented and tested it in an existing data flow analysis toolkit.

Keywords: Security-critical software; Data flow analysis; Taint analysis; Embedded programs; Downgrading

1 Introduction

Data flow analysis is a method of examining a system and how its constituents interact with each other. These interactions are embodied in the *flows* of data contained within the system, with the meaning of a flow depending on the type of system being analysed: for hardware it may represent the path of an electrical signal, while for software it may show that a particular value influences the output of a computation. Founded on the lattice model of secure information flow (Denning 1976), secure data flow analysis makes use of this technique to enforce or verify a system's security. Static analysis of data flow through security-critical program code, sometimes called 'taint' analysis, is relevant to protecting both data confidentiality and data integrity (Pistoia et al. 2007).

Several secure data flow analysis systems targeting software can be found in the literature (Suh et al. 2004, Newsome & Song 2005, Myers 1999, Song et al. 2008), but they all share a significant problem: the security classes of expressions are determined entirely from the security classes of their inputs; the expression's behaviour is not taken into consideration. This problem stems from the lattice model of secure information flow (Denning 1976) where the security class of a function $f(a_1, \ldots, a_n)$ is equal to the least upper bound of the input security classes, i.e., the least security class greater-than or equal-to all input security classes for parameters a_1 to a_n. This definition means that *downgrading* expressions (Li & Zdancewic 2005)—through which data may safely flow from a high-security domain to a low-security domain—cannot be recognised, leading to false-positive errors (Newsome et al. 2009) in which the analysis system states that a permissible flow is impermissible. For example, the security class of expression '`secret * 0`' is equal to that of classified variable `secret`, even though we can learn nothing about this integer's value from the expression's constant output.

Data flow analyses are important for information security, or 'infosec', evaluations of the kind performed by Australia's Defence Signals Directorate in its Australasian Information Security Evaluation Programme[1]. This programme evaluates information technology products against the *Common Criteria for Information Technology Security Evaluation* (ISO 2009) so that consumers of security-critical hardware and software can make informed procurement decisions, especially for government or military purposes. Most of these evaluations concern *embedded systems*, i.e., machines with limited resources, running software whose primary purpose is to interact with the physical world (Lee 2002). To accommodate these resource constraints, embedded software makes frequent use of bitwise operations to pack and unpack values from sub-byte sections of memory, meaning the security classes of data are often determined at the level of individual binary digits. Embedded program code analyses are thus particularly susceptible to false-positive errors because downgrading expressions may go unrecognised.

The work of 'infosec evaluators' of embedded systems can be supported by automated tools that identify potential data flow paths worthy of detailed investigation. One such tool is SIFA, the Secure Information Flow Analyser (McComb & Wildman 2005), in

This research was funded in part by the Defence Signals Directorate and the Australian Research Council via ARC Linkage-Projects Grant LP0776344.

[1] http://www.dsd.gov.au/infosec/aisep/

which graph representations of security-critical electronic circuitry may be developed and analysed in various ways (McComb & Wildman 2007), especially to identify data flow paths. As an extension to this tool, we recently produced a 'C-to-SIFA Converter', a compiler-like tool that produces SIFA-compatible data flow graphs of C code running on embedded microcontrollers. This tool, combined with SIFA, provides automated support for data flow analyses of both the hardware and software of security-critical embedded devices (Mills et al. 2012).

However, the data flow analyses performed by the C-to-SIFA Converter follow the traditional approach of treating each operation in a program code expression as atomic, thus failing to recognise potentially-downgrading expressions (Li & Zdancewic 2005). Our aim here, therefore, is to reduce the number of false-positive data flow paths generated during analysis of embedded software. To achieve this we (1) developed a library of detailed data flow models of C operators for different data representations, (2) devised an algorithm for inserting these models into a given data flow graph, (3) implemented the library and algorithm as an add-on module for the C-to-SIFA Converter, and (4) used case studies to confirm that the new module improved SIFA's ability to trace data flow through security-critical embedded software.

2 Related and previous work

Our concern is with analysing the way classified data may be transferred through expressions in embedded program code, with the aim of recognising expressions that successfully downgrade classified data. Here we review related and previous work from the areas of secure data flow analysis, program code certification, and embedded system certification.

2.1 Secure data flow analysis

In general, data flows may be identified either by examining a system's construction—*statically*—or by observing its behaviour—*dynamically*. Here we are interested in static analysis only; information about dynamic analysis for protecting data security can be found elsewhere (Suh et al. 2004, Newsome & Song 2005).

Many practical static data flow analysis systems are presented in the literature (Myers 1999, Song et al. 2008) but unfortunately they all generate false-positive errors (Newsome et al. 2009). Our goal is to help reduce this problem by developing a technique to precisely trace data flows through expressions, allowing downgrading expressions, i.e., those that produced unclassified outputs from classified inputs (Li & Zdancewic 2005), to be recognised.

Secure data flow analysis is founded upon the lattice model of secure information flow presented by Denning (1976). *Explicit* and *implicit* flows of data are differentiated, with the former occurring when data is transferred directly from one object to another, as in assignment statements, and the latter when execution of an explicit flow is controlled by another expression, such as when an assignment occurs within a conditional statement. For program code analysis Volpano, Smith & Irvine (1996) express this concept as a type-inference system and mathematically prove its soundness as a form of *noninterference* (Goguen & Meseguer 1982). These principles can be implemented as a static program code analysis (Pistoia et al. 2007) to trace the flow of security-critical data.

Importantly, however, *static* analyses can only identify *potential* data flows, many of which may never actually occur at run time. They thus overestimate the flow of data, producing false-positive results which waste the security evaluator's time and effort, motivating our interest in more precise analysis of code expressions.

In closely related work, Newsome et al. (2009) introduced a concept of *influence* which quantifies "how much" of a computation's inputs affect its output. They propose that security policies make use of this property, providing the example rule of "data influenced by x or more bits by the network may not be loaded into the program counter." However, this concept is also susceptible to false-positive errors because it is not known exactly *which* bits of the output are affected by the inputs; this makes it impossible to trace the original inputs through subsequent operations that remove data. For example, if we know that a value x has 4 bits of influence over another value y, and half of y is removed, it cannot be determined how many bits of x remain. Our goal is instead to trace the flow of individual bytes or even bits precisely.

Another highly relevant area of previous work is research into how information can flow through arrays in high-level languages. For instance, it has been observed that as well as an array's contents, its length and the indices at which certain values appear in the array can encode information (Deng & Smith 2004). More importantly, there has also been research into how to accurately trace the flow of classified information through specific array elements (Rus et al. 2007), rather than treating the array as atomic, which causes insertion of a single classified element to 'taint' the whole array. This research is obviously closely related to our own since all variables in computer programs can be modelled as arrays of bits. In fact, the problem we face in analysing *embedded* software is simpler than the general case for arrays because embedded code expressions make frequent use of hardwired constants, so we can often tell statically exactly which bits are affected by an operation, while this is extremely difficult for symbolic array indices (Rus et al. 2007).

2.2 Program code security certification

Denning & Denning (1977) build on the lattice model of permissable data flows (Denning 1976) to present mechanisms for certifying that a program is secure based on the flows of data within it; if it contains any impermissible flows, it could potentially leak sensitive information and is thus insecure. They extend the lattice model to include arrays, exceptions, and procedure calls so that these may also be certified through static analysis of program source code.

Several static and dynamic secure data flow analysis systems can be found in the literature. One such system is JFlow (Myers 1999), an extension to the Java language that adds a number of constructs enabling compile-time static analysis. Another system, BitBlaze (Song et al. 2008), supports both static and dynamic analysis of binary programs providing they use one of the several instruction sets supported by the tool. In our own research we have developed a static analysis tool for tracing data flow through security-critical C code in embedded devices (Mills et al. 2012).

However, all of these systems exhibit the same problem discussed in the previous section. They follow traditional data flow principles in which operations in expressions are treated as atomic, which results in false-positive errors because operations which

block classified data are not recognised as such. To partially solve this problem, Suh et al. (2004), Newsome & Song (2005) and Song et al. (2008) identify a set of "constant functions" whose inputs do not affect their output, providing as an example the instruction '`xor eax, eax`' which zeros the `eax` register. If given security-sensitive inputs, these constant functions become downgrading expressions. It is claimed that all three systems recognise a subset of these functions, but the authors do not explicitly state which are supported. Newsome & Song (2005) note that identification of these functions would greatly reduce the number of false-positive errors generated; this is also our objective, but our research aims to go further by also recognising operations that let *some* data through.

2.3 Embedded system certification

A special case of program code certification involves analysis of security-critical *embedded* code, i.e., software which interacts directly with its hardware environment. This is important in the context of international standards, such as the *Common Criteria for Information Technology Security Evaluation* (ISO 2009), which mandate information security, or 'infosec', evaluations of electronic devices intended to safeguard data in government and military applications. Within Australia the Defence Signals Directorate follows such standards to produce a list of trustworthy devices, known as the *Evaluated Products List*[2]. Our own research concerns the need to support such evaluations by automating data flow analysis of embedded software in the context of its surrounding digital circuitry (Mills et al. 2012).

This is challenging because embedded software typically contains bit-level and byte-level data operations not normally encountered in application software. To overcome memory constraints, embedded software often makes use of bitwise operations to pack and unpack data values. These operations may serve as downgrading expressions, by masking or otherwise eliminating security-critical bits, making embedded software analyses particularly susceptible to false-positive data flow errors if the function of these operations is not modelled accurately.

Previous research on automating data flow analysis of embedded devices produced SIFA, the Secure Information Flow Analyser, an open-source[3] software tool developed to automate data flow analysis of digital circuitry (McComb & Wildman 2005). SIFA represents electronic circuitry as a graph of *components*, each of which has a number *ports* on its periphery. Inter-component connections can be made by linking ports on different components, and intra-component data flow is modelled by defining how data is transferred between ports on the same component. SIFA provides a variety of graph-theoretic analysis functions (McComb & Wildman 2007), the most important being its ability to identify all data flow paths between selected ports, usually from a high-security data source to a low-security data sink.

In our own research we have developed a 'C-to-SIFA Converter', a compiler-like static analyser that converts embedded C code to SIFA-compatible data flow graphs that can be integrated into circuitry models. This capability supports seamless data flow analyses through an embedded device's hardware and software (Mills et al. 2012). However, the tool follows standard data flow graph building principles in which each operator in an expression is represented as an atomic component and each program variable or constant is modelled as a single arc (Scholz et al. 2008).

Thus, our tool also suffers from the inability to recognise expressions that downgrade classified data at the bit or byte level. Our goal here, therefore, was to extend the capabilities of the C-to-SIFA Converter to give it a more precise ability to analyse data flow through expressions in embedded C code.

3 The component library

The first part of our solution was to develop a SIFA-compatible library of data flow components that model the way data flows through operations in C expressions precisely. This section discusses the rationale for the library's design and presents some representative component models.

3.1 Threat model

In designing the library we had to first decide precisely what was meant by 'data flow'. The ways in which *information* can flow through program expressions can be subtle. For instance, the value of expression '`A + B`' tells us nothing about the values of operands `A` and `B`. However, the value of expression '`A * B`' always reveals whether or not the signs of the operands were the same. Furthermore, if the expression's value is a prime number then we know the exact value of the operands! Rather than entering into complex deliberations about what information can be inferred from expression values, we therefore instead chose to adopt the simple and clear *noninterference* model of data flow cited above. Thus, unless it can be proven otherwise, we conservatively assume that the values of an expression's operands all exert some discernable influence on the expression's value. Thus, both operands `A` and `B` are assumed to affect the value of expression '`A + B`', even though little or nothing can be learnt about their precise values without additional information, but the value of expression '`A * 0`' is assumed to reveal nothing at all about operand `A` since constant `0` entirely dominates the result.

This overall argument is based on a scenario in which the 'attacker' of our program has access to its source code listing and can observe all data emanating from the embedded device containing the microcontroller running the compiled code. This is a worst-case attacker profile for an embedded security-critical device, so our model is thus safely conservative, even though it will sometimes overestimate data flow. (If the attacker had powers greater than this, e.g., physical access to the device and the ability to insert debugging probes into the microcontroller to observe memory or register values, then no defence is possible.)

3.2 Design process

In order to accurately trace data flow through the expressions that the C-to-SIFA Converter encounters, we must first understand how the component parts of these expressions behave in isolation. As the tool targets embedded C code, there are several kinds of expression element, including but not limited to:

- bitwise, logical, and mathematical operators such as `<<`, `&&`, or `+`,
- calls to built-in and user-defined functions such as `abs()` or `rand()`,

[2] http://www.dsd.gov.au/infosec/epl/
[3] http://sifa.sourceforge.net/

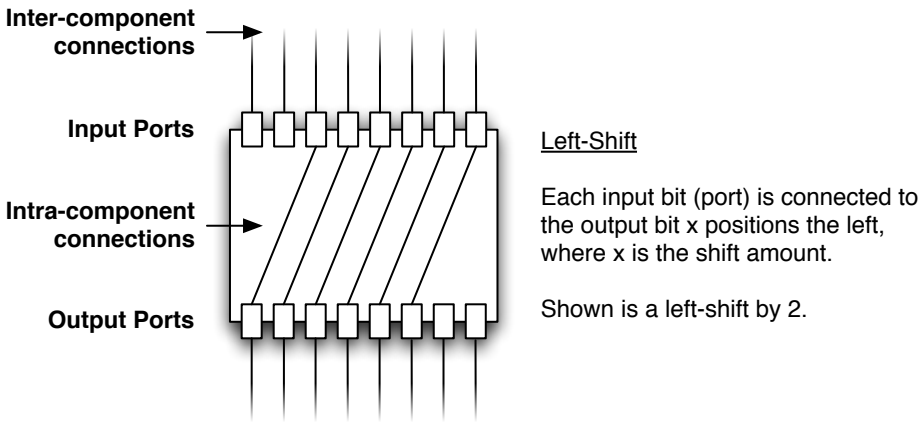

Figure 1: A component depicting a left-shift by 2.

- identifiers and variables such as `PI` or `x`,
- literal values of varying types such as `42` or `'c'`, and
- type casts such as `(int)` or `(float)`.

We considered the behaviour of a significant number of these expression elements under various circumstances. These ranged from different input types (e.g., floating-point values) to specific sets of input values (e.g., powers of two). The results of these studies were modelled as SIFA *components* which may be connected to form data flow graphs modelling entire expressions. Particular emphasis was put on identifying expression elements that remove data as they have the potential to be used in downgrading expressions (Li & Zdancewic 2005).

The C-to-SIFA Converter targets the CCS Embedded C compiler, so this compiler's reference manual[4] was used to determine which expression elements should be analysed, and to gain a basic understanding of their behaviour. A number of other works were then consulted to further develop our understanding of the elements' behaviour (Tanenbaum et al. 1982, Seacord 2006). The expression elements were then translated to SIFA components as explained in Section 3.3 below.

Given the vast number of C expression elements that the tool may encounter, we could model only a subset of them in the time available for this research project. We were guided in our choices by some C program fragments supplied to the research team by the Defence Signals Directorate as examples of 'typical' software found in security-critical embedded devices undergoing analysis in the Australasian Information Security Evaluation Programme. We considered all bitwise, logical, and mathematical operators, but excluding assignment operators such as '`+=`' and '`|=`' because they can be expanded into two simpler operations. A subset of standard C library functions and type casts were also considered. These are the expression elements most frequently used in embedded C code encountered by AISEP security evaluators, and are thus the most likely to cause false positives.

3.3 Expression components

A *component*—an entry in the component library—describes the behaviour of a particular expression element under some circumstance, often for specific input values. The data flow components are represented in SIFA's graphical notation as a named collection of 'ports'. (In fact, SIFA's graph analysis algorithms treat a model's ports as the graph's nodes, rather than the components.) An example library component for a left-shift by 2 on an 8-bit value is shown in Figure 1.

A component may have any number of input and output *ports* that represent data flowing into and out of it, respectively. These ports also act as "connection points" from which inter- or intra-component connections may be made. Inter-component connections join the output ports of one component to input ports of another; it is this type of connection that allows entire expressions to be modelled using components. Intra-component connections join input and output ports of a single component and model the flow of data through it. Both types of connection depend on the component's input and output *representations* (Section 3.4), but intra-component connections also depend on the behaviour of the expression element being modelled.

To determine what intra-component connections should be made through a component, we followed the concept of *noninterference*, originally defined by Goguen & Meseguer (1982). Volpano et al. (1996) provide an alternate definition that is more suited to our work as it concerns memory and the variables within it as opposed to the original definition which considers users and the actions that they may perform. Using this concept, if an input port is noninterfering with all output ports, it does not affect the component's output and is said to be "removed." If an input port may affect an output port however, an intra-component connection is made between them. This means that there are situations where an input port may be connected to several output ports or vice versa. For example, when performing addition a carry will occur if two bits of equal significance are both 1, which may cause another carry and so on.

Literal constant inputs are not compatible with this definition of removal as they are immutable in the context of static data flow analysis: they cannot be changed without modifying the program's source code. It is for this reason that entries in the component library do not include input ports for literal values. Were they to be included, they would serve only to complicate the data flow graph and would provide no benefit as it is assumed that all "noteworthy" constants (e.g., the seed of a pseudorandom number generator used to create encryption keys) will be named in the source program. (This particular issue was much discussed during development of the C-

[4] http://www.ccsinfo.com/

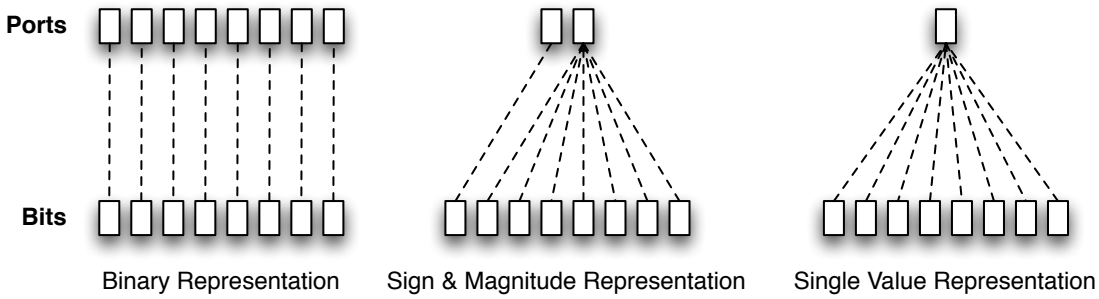

Figure 2: Three of the data representations used in the component library, showing how the individual bits of a value (bottom) are mapped to ports in the model (top).

to-SIFA Converter. Ultimately it was decided that information security evaluators would not usually want to trace the 'flow' of a hardwired constant, although some provision for this is built into the tool.)

3.4 Data representations

At times it is convenient to explain the behaviour of an expression element in terms of logical aspects of values rather than the underlying bits of memory. For example, it is clearer to say that the absolute-value function `fabs()` removes the sign of its operand, rather than saying the leftmost bit is discarded. Furthermore, to minimise the complexity of the data flow graphs, we want to avoid tracing every individual bit through an expression in situations where this does not provide any helpful information.

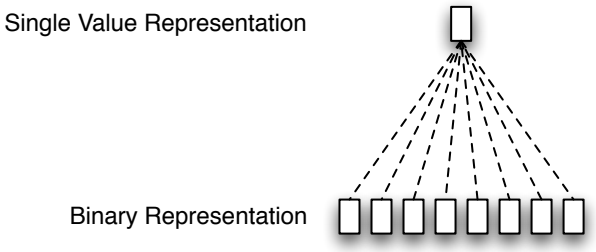

Figure 3: An undesirable connection between two representations in which a single value (top) connects to all bits in a binary representation (bottom).

To do this, the inputs and outputs of all components in the component library are expressed in particular *representations* that map the underlying bits of values to components' ports. By mapping certain groups of bits to a single port, components can effectively operate on logical aspects of values; this lets us create components for expression elements whose behaviour would otherwise be prohibitively complex to analyse. Three of the representations used in the component library are shown in Figure 2:

- binary representation where each bit is mapped to its own port,
- sign and magnitude representation where the sign bit is mapped to one port and the remaining bits (the magnitude) are mapped to another, and
- single value representation where all bits are mapped to a single port.

As representations simply map bits to ports, we can make inter-component connections between any two representations while maintaining the integrity needed for data flow analysis. Providing the representations have the same underlying data type, this is achieved by iterating over all bits of the data, and connecting the ports that represent each bit.

Connections between some representations are undesirable, however, as they result in an apparent duplication of data that undermines the integrity of the data flow analysis. This occurs when the first representation maps two or more bits to a single port, and the second representation maps those same bits to a greater number of ports. This can be seen in Figure 3 where a single value representation is connected to a binary one. Connecting data representations in this way is unhelpful because bitwise data flow cannot be traced through the single value representation.

3.5 Some typical library components

The component library contains data flow components for arithmetic operators, comparison and relational operators, logical operators, bitwise operators, type conversions and standard library functions. For each operator there may be several data flow components for different representations and for different compile-time constant operands. There are far too many components to show here, but a few of the more interesting examples are described below. In each case `A` and `B` denote variables whose values are not known statically, and x and y are integers used to denote unspecified parts of a compile-time constant, e.g., 2^x denotes any integer that is a power of 2. (These values are part of the component definition, not necessarily the expression.)

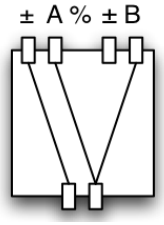

Figure 4: Library component for modulus expression '`A % B`' using sign-and-magnitude representation.

For instance, Figure 4 shows the component for sign-and-magnitude representation of the modulus operator. In this case the magnitude of the result (at the bottom of the figure) is affected by the magnitudes of both operands, but the sign of the result is affected only by the sign of the first operand. Thus, this operation would successfully downgrade an expression in which only the value of the second operand's sign was classified.

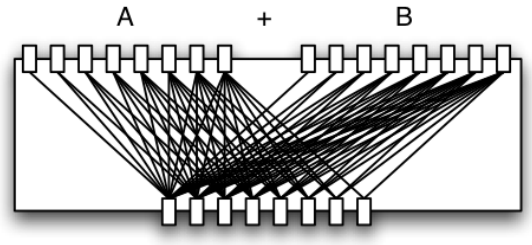

Figure 5: Library component for addition '`A + B`' using binary representation.

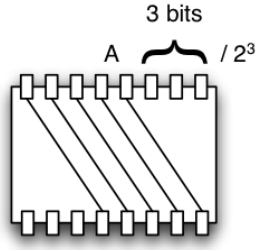

Figure 6: Library component for unsigned integer division '`A / `2^x', where x is 3, using binary representation.

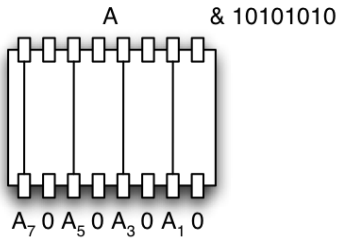

Figure 7: Library component for bitwise conjunction '`A &` x', where constant x's value has the bit pattern shown on the right, using binary representation.

A more complex component is for the binary representation of integer addition, as shown in Figure 5. In this case each input bit affects all outputs of equal or higher significance, because the addition of any two bits may cause a carry.

The component in Figure 6 models a special case of unsigned integer division in which the second operand is a compile-time constant and a power of 2. In effect, this operation is equivalent to a right shift, '`A >>` x', so it may downgrade input `A` by removing its rightmost x bits. (This particular component does not apply to signed values.)

Obvious cases of downgrading often occur with bitwise operations. For instance, Figure 7 shows the result of performing a logical 'and' operation on a byte `A` and a compile-time constant bit pattern. In this case bits in `A` that correspond to zeros in the second operand are downgraded.

A less-obvious example is the less-than-or-equal comparison in Figure 8. In this case the second operand is a constant whose value can be expressed in the form $2^x - 1$. Since the x least-significant bits of the first operand `A` cannot sum to more than $2^x - 1$ they cannot influence the result and hence are effectively downgraded.

Many components have no data flow from their input(s) to their outputs at all. Most notably, the value of expressions that produce a constant value do not

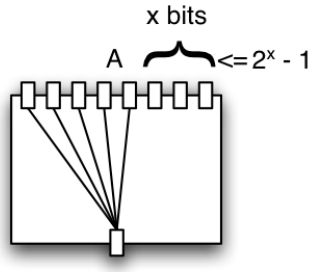

Figure 8: Library component for integer comparison '`A <=` $2^x - 1$' using binary representation.

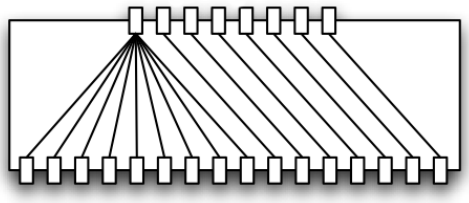

Figure 9: Library component for type conversion '`(`T`)A`', where target type T has a greater range than that of signed integer `A`, using binary representation.

reveal anything about the values of their operands. Some examples include '`A * 0`', '`A % 1`', '`A == A`' and '`B >=` x' where `B` is an m-bit two's complement integer and x is the minimum such value -2^{m-1}. The library also contains components modelling the behaviour of expressions involving the special C floating point value 'not a number', many of which also return constant results.

Finally, not all operators of security relevance in the library involve blocking data. In a few instances, such as that shown in Figure 9, classified data may be duplicated. In this case a signed integer is cast to a type with a larger range (e.g., 'short' to 'long') in which case sign extension is applied and the sign bit is copied multiple times.

4 Expression tracing functions

Having defined the library of data flow components, the second challenge was to use them in the data flow graphs generated for embedded C code expressions by the C-to-SIFA Converter. In this section we explain how this tool was extended, including the algorithms for selecting library components and for connecting them together.

4.1 Rationale

As mentioned previously, we developed the C-to-SIFA Converter to support the work of AISEP security evaluators (Mills et al. 2012). The tool automates the creation of data flow graphs for embedded software, taking embedded C source code as input, and constructing a data flow graph in a format readable by SIFA (McComb & Wildman 2005). The tool itself is written in C♯ 3.0 and targets embedded C code that compiles under the CCS compiler.

At the start of this research the C-to-SIFA Converter followed the lead of other secure data flow analysis systems and treated expression operators as atomic, resulting in false-positive errors, i.e., connections through the graph in situations where a more detailed analysis would show that there is no data

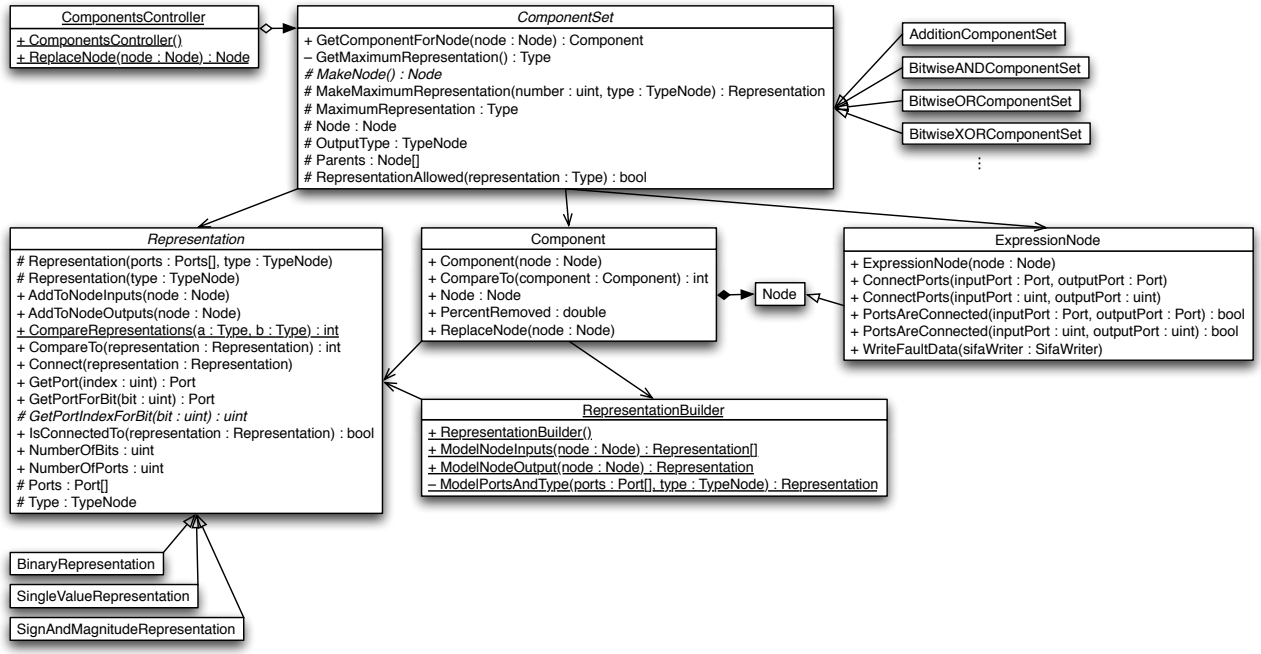

Figure 10: Classes developed to implement the expression tracing functionality.

flow. Our new expression tracing functionality aimed to resolve this issue by replacing atomic components with more accurate ones sourced from the component library.

Being an extension to the existing C-to-SIFA Converter, the expression tracing functionality was also written in C♯ 3.0. The set of classes and methods developed is shown in Figure 10. To help guarantee the program's correctness, test-driven development was utilised, with the final test suite containing 125 unit tests. The `ReplaceNode()` method of the `ComponentsController` class acts as the interface to the expression tracing tracing functionality, taking a `Node` from the data flow graph (an abstraction of a SIFA component) and replacing it with a more accurate one sourced from the component library. This process can be divided into two distinct stages: selection of the replacement, and replacing the existing `Node`. Detailed discussions of these two stages are provided in Sections 4.3 and 4.4, respectively.

4.2 Representations

Representations are modelled in the expression tracing functionality by the abstract `Representation` class and its subclasses. Methods are provided to, among other things, retrieve the `Port` representing a particular bit and connect arbitrary `Representation` instances. Each subclass implements the protected `GetPortIndexForBit()` method which effectively provides the "subclass-specific" logic used in the aforementioned functions. Three subclasses are defined in the expression tracing functionality:

- `BinaryRepresentation` where each each bit is mapped to its own port,
- `SignAndMagnitudeRepresentation` where the sign bit is mapped to its own port, and all other bits (the magnitude) are mapped to another, and
- `SingleValueRepresentation` where all bits are mapped to a single port.

A `Representation` instance can be created in one of two ways: it may be instructed to create its own `Port` array, or it can use an array sourced from an existing `Node`. The former method is used when creating a new `Node`, while the latter is used when creating a `Representation` for an existing one. Each subclass implements constructors for both of these methods so that they may calculate the number of required ports or validate those that were given.

Making use of the second method is the `RepresentationBuilder` class which provides functions to create `Representation` instances modelling input or output ports of an existing `Node`, `ModelNodeInputs()` and `ModelNodeOutputs()`. As a component may have several logical inputs, the former method returns an array, while the latter returns a single object. This functionality is used extensively when replacing `Node`s, with `Representation`s being built for all parents and children, and then connected to the new `Node` using `Representation.Connect()`. The `RepresentationBuilder` functions work by firstly identifying all subclasses of `Representation` that could validly be used at this point in the data flow graph, as per the principles in Section 3.4 above, and by then selecting the most precise representation available.

4.3 Component selection

The `ComponentSet` class hierarchy is the result of translating the component library to code, with each subclass representing a logically related group of components. These components may be accessed by calling the `GetComponentForNode()` method which returns the most suitable replacement for the given `Node`, or `null` if none are available. To find all potential replacements for a particular `Node`, this method is firstly called on all `ComponentSet` subclasses. This design was chosen as it allows logically related components to be grouped together in the code, but leaves the nature of that relation up to the developer. In the expression tracing functionality, components are grouped by

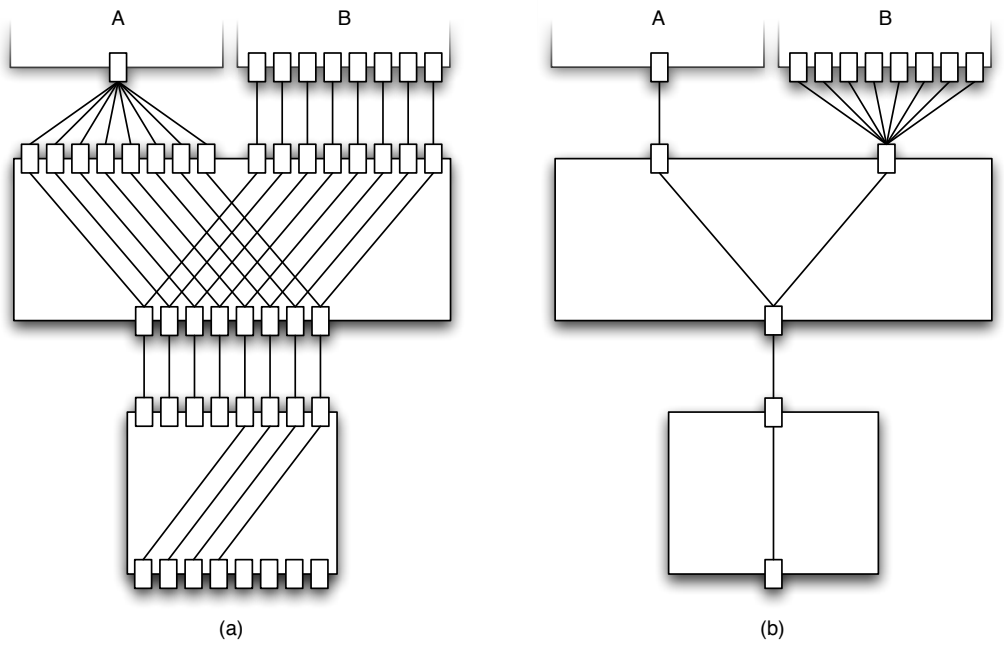

Figure 11: Data flow through expression `(A & B) << 4` modelled using different representations.

the expression elements that they model, resulting in subclasses such as `AdditionComponentSet` and `BitwiseANDComponentSet`. As intra-component connections may not be set on instances of the base code's `Node` class—it assumes that all input ports are connected to all output ports—replacement components are constructed using its `ExpressionNode` subclass.

The `ComponentSet` class offers a number of "helper" methods to support the creation of replacement components in its subclasses. One such method is `RepresentationAllowed()` which returns a boolean value indicating whether the replacement component may use the given `Representation`, enforcing the constraint on connections from less- to more-precise representations as discussed in Section 3.4. However, this method differs slightly from the original definition when the `Node` that is to be replaced has multiple parents with varying output representations. In this situation, the replacement component may use the most-precise of the parents' output representations to ensure that removal of that parent's data is recognised.

This is demonstrated in Figure 11 where the bitwise **and** component uses (a) the most-precise of the parents' output representations (binary) and the four most-significant bits of `B` are removed, as opposed to (b) the least-precise representation (single value) where this removal is not recognised. The `GetMaximumRepresentation()` method calculates and returns the most-precise of the parents' output representations, or simply returns `BinaryRepresentation` when there are no parents.

Once the set of potential replacement components has been compiled, the best of these must be selected. As the goal of this research is to better recognise expressions that remove data, the best component is the one that removes the greatest percentage of its input ports. Percentage removed is used over number removed as the significance of a single port varies between representations; it is better to remove a port in single value representation than in binary representation. This process is aided by the `Component` class's implementation of the `IComparable<Component>` interface, allowing the potential replacements to be sorted by percentage of input ports removed. Should multiple components remove the same percentage of their input ports, the component with the most-precise representation is chosen.

Unlike entries in the component library, components constructed by `ComponentSet` subclasses include ports for literal value inputs. These ports are not involved in any intra-component connections, however, making the components effectively equal to those of the component library. This approach was taken as the C-to-SIFA Converter's base code already inserts literal value nodes into the data flow graph, and it allows for the future addition of an option to trace literal values of interest, i.e., "magic numbers."

As the component selection algorithm is affected by the output representations of a node's parents, the order in which nodes are replaced is significant. If a child node is replaced before its parent, the most appropriate component may not be selected as the parent's output will be in single value representation. To ensure that the most appropriate components are used throughout the entire data flow graph, replacement should start at nodes without parents (e.g., literal value nodes or those for the `rand()` function) and follow on those encountered while performing a breadth-first graph traversal from the same node.

4.4 Component replacement

Once an appropriate replacement has been found, the existing `Node` must be removed from the data flow graph and its replacement inserted. This is a two step procedure. First, all connections involving the existing `Node` are mirrored with its replacement. Supporting this process is the `RepresentationBuilder` class which provides methods to construct `Representation` instances for existing `Nodes` and thus allows them to be connected, as was discussed in Section 4.2. The second step is to sever all connections to the existing `Node`, removing it from the data flow graph.

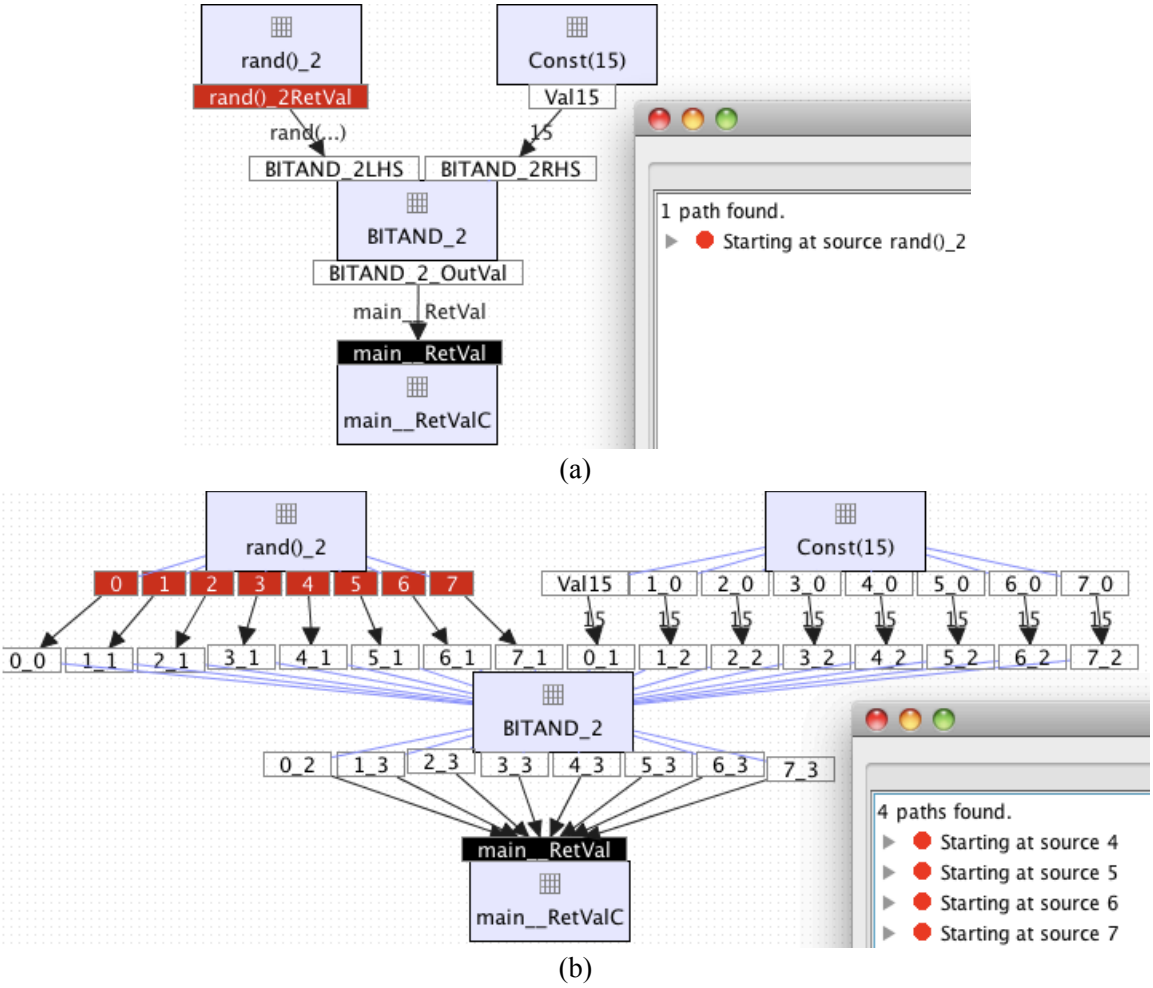

Figure 12: Data flow graph for expression '`rand() & 0x0F`' in (a) the original and (b) the updated tool.

5 Evaluation

In this section we demonstrate the effectiveness of our extension to the C-to-SIFA Converter's expression modelling functionality on some small case studies involving expressions or program segments cited in other works or that the tool was known to have difficulty with. If the updated version of the tool arrives at the same conclusion as the previous works or overcomes the difficulties it previously faced, it can be said to have greater accuracy and will result in fewer false-positives.

For each case study, an equivalent C program was developed and passed to both the original and updated versions of the C-to-SIFA Converter. The resulting data flow graphs were opened in SIFA and analysed using its "find all paths" function which enumerates all paths between selected "source" and "sink" ports. If a path exists between two ports, data is assumed to flow between them. In the SIFA screenshots in Figures 12 to 14 the source and sink ports appear in red and black, respectively, as is customary in the infosec community.

5.1 Case study: Masking using bitwise 'and'

After introducing the concept of influence, Newsome et al. (2009) provide several examples to demonstrate its behaviour and use. One such example is the assignment '`V = base + (I & 0x0F)`', for which the authors conclude that variable `I` has 4 bits of influence over variable `V`. This example demonstrates that the expression's bitwise **and** operation removes all but the 4 least-significant bits of `I`.

Our C program mirroring this example performs the same masking operation, but instead uses the output of the `rand()` function because the value of `I` in the expression is unknown. As can be seen at the top of Figure 12, the original version of the C-to-SIFA Converter treated the bitwise **and** operation as atomic. When SIFA was used to analyse data flow it merely reported that the `rand()` function's value affected the expression's output. The updated tool, however, recognised that a bitwise component could be used in this situation. As shown at the bottom of Figure 12, it expanded the values into binary representation and arrived at the same conclusion as Newsome et al. (2009), with SIFA now reporting that only the 4 least-significant bits of `rand()` pass through the bitwise **and** operation. The more detailed data flow graph thus allows us to see which parts of the value are effectively blocked.

5.2 Case study: Dropping bits via integer division

Another example of influence provided by Newsome et al. (2009) is the assignment '`V = I / 2`', for which the authors conclude that variable `I` has control over all but the most-significant bit of variable `V`. This example demonstrates that division by 2^x for $x \geq 0$ is equivalent to a right-shift by x places. In this case, x equals 1.

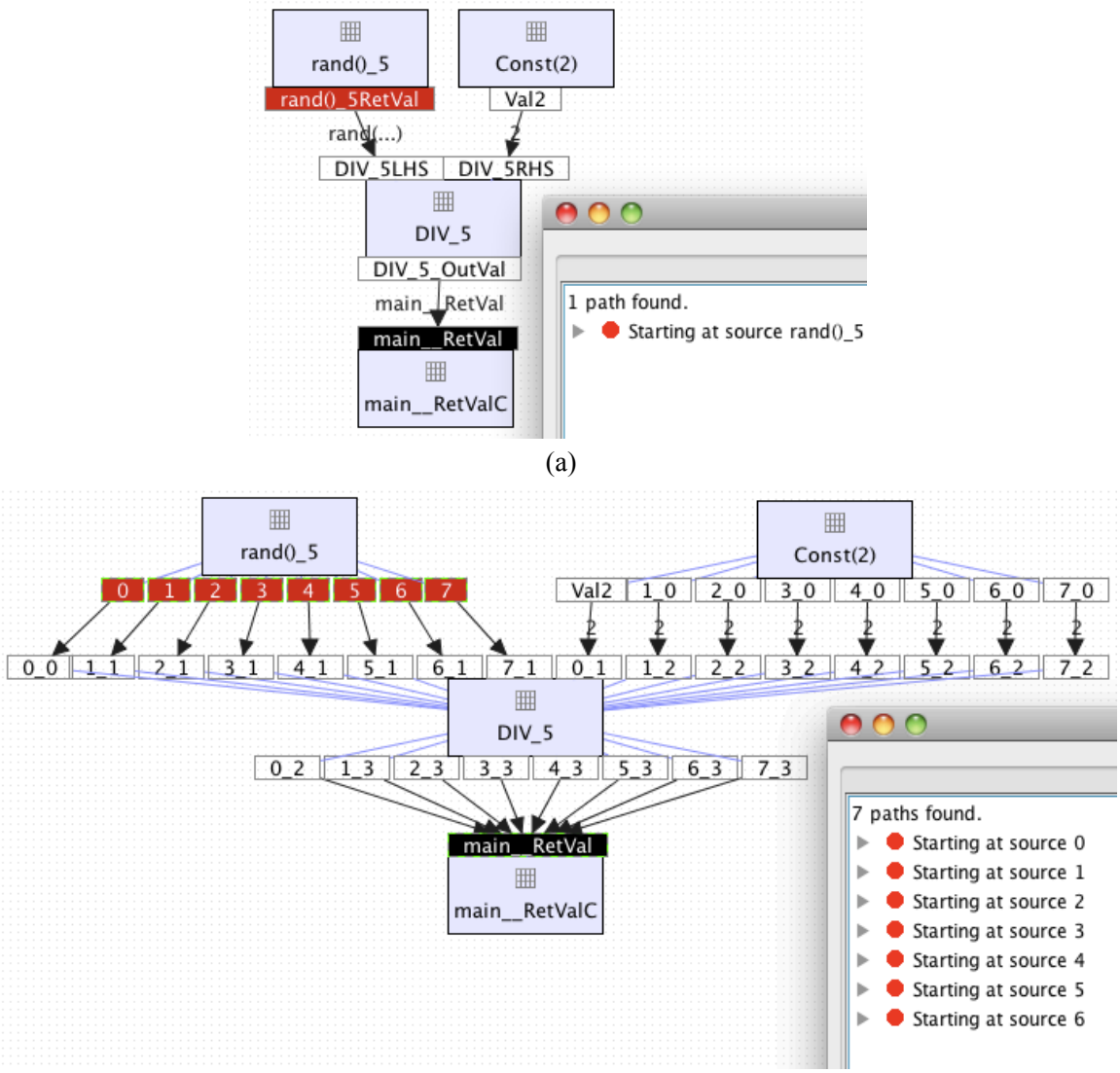

Figure 13: Data flow graph for expression '`rand() / 2`' in (a) the original and (b) the updated tool.

The C program used to mirror this example performs the same division operation but again on the value of the `rand()` function, for the same reasons as in the previous case study. As can be seen at the top of Figure 13, the previous version of the tool again treats the operation as atomic and SIFA merely reports that the `rand()` function's value affects the whole expression's value. However, the bottom of Figure 13 shows that our extended C-to-SIFA Converter again expanded the values into a binary representation, and inserted an appropriate division component, allowing SIFA's data flow analysis to recognise that the least-significant bit of the value produced by the `rand()` function is removed by the division operation. Again this accords with the assessment of this expression by Newsome et al. (2009).

5.3 Case study: Ineffective bits in a comparison

To help us understand how the C-to-SIFA Converter could be used by AISEP security evaluators, the Defence Signals Directorate provided this project with a package of code fragments intended to be representative of the sort of embedded C code they encounter during their evaluations. One of these code fragments contains the expression '`source_reg > 0x7F`' which effectively removes the 7 least-significant bits of `source_reg` as they alone cannot make the value exceed `0x7F` (127); it is the most significant bit that determines the expression's output. This demonstrates that relational operators may be used to downgrade data similarly to bitwise operators.

Again, the C program used to mirror this example uses the `rand()` standard library function in place of the unknown value `source_reg`. Once more the top of Figure 14 shows that the previous version of the tool treats the greater-than operation as atomic and provides no detail about how data flows through it. However, the bottom of Figure 14 shows how the updated tool expands the values into binary representation and correctly removes the 7 least-significant bits of `rand()`, allowing only the most-significant bit to pass through.

6 Discussion and future work

This research project was time-constrained, so we did not attempt to create library components for all of those expression elements supported by the CCS com-

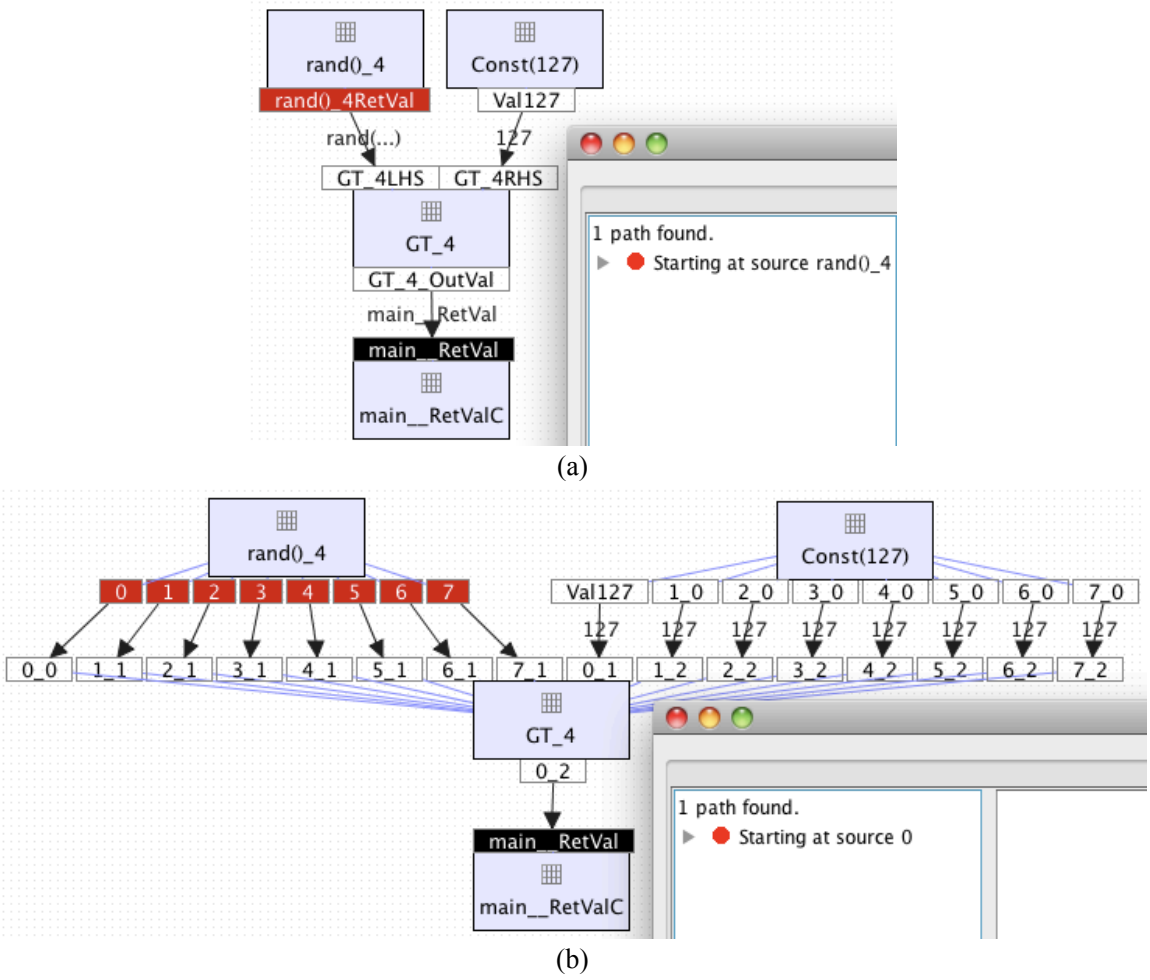

Figure 14: Data flow graph for expression '`rand() > 0x7F`' in (a) the original and (b) the updated tool.

piler. Instead we considered certain bitwise, logical, and mathematical operators, as well as a subset of standard C library functions and type casts, that were found in the package of code fragments provided to the project by the Defence Signals Directorate as representative of the sort of embedded C code encountered during infosec evaluations

The accuracy of the new expression tracing functionality could be further increased by creating support for more expression elements such as operations that involve the special floating-point values `Inf` (infinity) and `NaN` (not a number). Better type cast support would be particularly beneficial as such operations are used frequently within embedded code and can potentially remove significant amounts of data.

While adding support for expression elements increases the tool's accuracy directly, adding representations does so indirectly as it facilitates the creation of components. Several other representations were contemplated during the development of the component library, but were not considered important enough to implement given the 'typical' code fragments provided by the DSD. They include:

- byte representation where groups of 8 bits are represented by a single port,
- sign, mantissa, and exponent representation where bits are grouped based on how floating-point values are stored in memory (e.g., as per the IEEE 754 standard), and
- sign, integral portion, and fractional portion representation (but we cannot determine which bits comprise these portions without knowing the value).

A nontrivial undertaking that could significantly increase the tool's accuracy would be to implement some symbolic execution functionality, e.g., by associating additional metadata with the data values being traced through components. For example, after data flows through a left-shift by x, we know that its x least-significant bits are 0; this would allow more appropriate components to be used later in the data flow graph, further reducing false-positive errors. This functionality could be implemented by developing an abstraction of data to which information can be attached, and then doing so as data is traced through the data flow graph.

Finally, the expression tracing extension to the C-to-SIFA Converter was constrained in some ways by the tool's existing code base. Data could not be traced through type casts, for example, because the extensions works by replacing nodes of a given data flow graph constructed by the base code, and the version of the C-to-SIFA Converter used in this project did not insert nodes for type casts. (The C-to-SIFA Converter is still being refined, however, so such a capability could be added later.)

7 Conclusion

In this research we have developed a technique to precisely trace data flow through embedded program

expressions and used it to reduce the overestimation of data flow made by an existing static-analysis toolkit intended for evaluating security-critical program code. Small case studies such as those shown above have demonstrated that the enhanced toolkit is more accurate and generates fewer false-positive errors through the increased recognition of downgrading expressions (Li & Zdancewic 2005). This has the potential to save time and effort for information security evaluators of embedded devices. At the time of writing we are conducting larger case studies (Mills et al. 2012) and further expanding the capabilities of the toolkit. As noted above, there are also a number of ways in which the expression tracing functionality described herein could be improved even further.

References

Deng, Z. & Smith, G. (2004), Lenient array operations for practical secure information flow, *in* 'Proceedings of the Seventeenth IEEE Computer Security Foundations Workshop (CSFW 2004), California, USA', IEEE Computer Society, Washington DC, USA, pp. 115–125.

Denning, D. (1976), 'A lattice model of secure information flow', *Communications of the ACM* **19**(5), 236–242.

Denning, D. & Denning, P. (1977), 'Certification of programs for secure information flow', *Communications of the ACM* **20**(7), 504–513.

Goguen, J. & Meseguer, J. (1982), Security policies and security models, *in* 'Proceedings of the 1982 IEEE Symposium on Security and Privacy', pp. 11–20.

ISO (2009), *ISO/IEC Standard 15408-1:2009, Information Technology—Security Techniques—Evaluation Criteria for IT Security—Part 1: Introduction and General Model*, 3.1 edn, International Organization for Standardization, Geneva, Switzerland. Standardised version of the Common Criteria for Information Technology Security Evaluation.

Lee, E. (2002), Embedded software, *in* M. Zelkowitz, ed., 'Advances in Computers', Vol. 56, Elsevier, pp. 55–95.

Li, P. & Zdancewic, S. (2005), Downgrading policies and relaxed noninterference, *in* 'Proceedings of the 32nd ACM Symposium on Principles of Programming Languages (POPL'05)', ACM, pp. 158–170.

McComb, T. & Wildman, L. P. (2005), SIFA: A tool for evaluation of high-grade security devices, *in* C. Boyd & J. Nieto, eds, 'Proceedings of the Tenth Australasian Conference on Information Security and Privacy (ACISP 2005), Brisbane, Australia', Vol. 3574 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 230–241.

McComb, T. & Wildman, L. P. (2007), A combined approach for information flow analysis in fault tolerant hardware, *in* 'Proceedings of the Twelfth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2007)', IEEE Computer Society Press.

Mills, C., Fidge, C. J. & Corney, D. (2012), Tool-supported dataflow analysis of a security-critical embedded device, *in* J. Pieprzyk & C. Thomborson, eds, 'Proceedings of the Tenth Australasian Information Security Conference (AISC 2012), Melbourne', Vol. 125 of *Conferences in Research and Practice in Information Technology*, Australian Computer Society.

Myers, A. (1999), JFlow: Practical mostly-static information flow control, *in* 'Proceedings of the 26th ACM Symposium on Principles of Programming Languages', ACM, pp. 228–241.

Newsome, J., McCamant, S. & Song, D. (2009), Measuring channel capacity to distinguish undue influence, *in* S. Chong & D. Naumann, eds, 'Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security (PLAS'09), Dublin, June 15', ACM, pp. 73–85.

Newsome, J. & Song, D. (2005), Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software, *in* 'Proceedings of the 12th Annual Network and Distributed System Security Symposium'.

Pistoia, M., Chandra, S., Fink, S. J. & Yahav, E. (2007), 'A survey of static analysis methods for identifying security vulnerabilities in software systems', *IBM Systems Journal* **46**(2), 265–288.

Rus, S., He, G. & Rauchwerger, L. (2007), Scalable array SSA and array data flow analysis, *in* E. Ayguadé et al., eds, 'Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2005)', Vol. 4339 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 397–412.

Scholz, B., Zhang, C. & Cifuentes, C. (2008), User-input dependence analysis via graph reachability, *in* 'Proceedings of the Eighth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008), Beijing, September 28–29', IEEE, pp. 25–34.

Seacord, R. (2006), *Secure Coding in C and C++*, Addison-Wesley. ISBN 0-321-33572-4.

Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M., Liang, Z., Newsome, J., Poosankam, P. & Saxena, P. (2008), BitBlaze: A new approach to computer security via binary analysis, *in* R. Sekar & A. Pujari, eds, 'Proceedings of the 4th International Conference on Information Systems Security', Vol. 5352 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, pp. 1–25.

Suh, G., Lee, J., Zhang, D. & Devadas, S. (2004), Secure program execution via dynamic information flow tracking, *in* 'Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems', pp. 85–96.

Tanenbaum, A., van Staveren, H. & Stevenson, J. (1982), 'Using peephole optimization on intermediate code', *ACM Transactions on Programming Languages and Systems* **4**(1), 21–36.

Volpano, D., Smith, G. & Irvine, C. (1996), 'A sound type system for secure flow analysis', *Journal of Computer Security* **4**(3), 167–187.